National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Services des thèses canadiennes

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r.86/06)

Canada

The University of Alberta

THE USE OF RECURSIVE TRANSITION NETWORKS FOR DIALOGUE DESIGN IN USER INTERFACES

by

Sai Choi LAU

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1986

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-30276-3

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Sai Choi LAU

TITLE OF THESIS: The Use of Recursive Transition Networks for Dialogue Design in User Interfaces.

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1986

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) .....

Permanent Address:
217, Azalea House,
So Uk Estate,
Kowloon,
Hong Kong.

Dated Feb, 1, 1986

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **The Use of Recursive Transition Networks for Dialogue Design in User Interfaces.** submitted by **Sai Choi LAU** in partial fulfillment of the requirements for the degree of **Master of Science.**

.....

Supervisor

.....

.....

.....

Date Oct. 10, 1985

ABSTRACT

There is currently great interest in the automatic generation of user interfaces to software systems. These generators are called User Interface Management Systems (UIMS). At the University of Alberta, a UIMS is being built based on the user interface model developed at the Seeheim Workshop on User Interfaces. This model divides the User Interface Management System into a presentation component, a dialogue control component and an application interface model. The dialogue control component is the control module of the system. Recursive Transition Networks (RTNs), context free grammars and event languages are the three main notations for describing the dialogue. An RTN describes the response of a system by means of state transition diagrams.

In this thesis, a graphical editor is described. This editor is used to enter the RTNs into the system interactively. The user interface designer enters the diagrams (RTNs) into the system in a graphical way and the data is stored in a database.

In order to provide a system that accommodates all three dialogue control notations and to facilitate implementation, all three notations are converted into a common format. Since an event-based model has greater descriptive power, a special event format, the "Event Based Internal Form", is designated as this common format. The second part of the thesis describes some conversion algorithms used to convert the RTNs (which have been entered by the editor) into this special format.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my supervisor, Dr. M. Green, for his guidance, assistance, criticisms and encouragement during the course of this project and the preparation of the thesis.

I am grateful for the careful reading and comments provided by the members of my examination committee, Drs. W.W. Armstrong, M.T. Ozsu and K.A. Stromsmoe.

Also, financial support of the computing Science Department in the form of teaching and research assistantships are gratefully acknowledged.

Lastly, special thanks to all my friends for their encouragements during my two years stay here.

Table of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. Descriptions of the UIMS	1
1.2. Purposes of the project	3
1.3. Organization of the Thesis	5
Chapter 2: Background	6
2.1. Description of the Dialogue Control Component	6
2.2. The Recursive Transition Network	8
2.3. The Event Model	10
2.4. Review of Previous Work	12
2.4.1. Newman's System	12
2.5. Summary of Existing Systems features	16
2.6. The University of Alberta UIMS Requirements	16
Chapter 3: Designing the Graphical Editor	18
3.1. Introduction	18
3.2. Basic Aims	18
3.3. Designing the User Interface	19
3.3.1. Potential Users	20
3.3.2. Minimal Memory Required	21
3.3.3. Immediate Feedback	22
3.3.4. User Model	23
3.4. Designing the Editor Commands	24

3.4.1. Basic Commands 24
3.4.2. Auxiliary Commands 25
3.5. Design of Manipulated Objects 26
3.5.1. Basic Objects 26
3.5.2. Auxiliary Objects 27
3.6. The Data Structure 29
3.7. Menu Layout 30
3.8. Error Handler 31
3.8.1. Operation Errors 31
3.8.2. Inconsistent Errors 31
3.9. Supporting Facilities 32
Chapter 4: Implementation of the Graphical Editor 33
4.1. Introduction 33
4.2. The Environment 33
4.3. Implementation of the User Interface 34
4.3.1. The Menus 34
4.3.2. Window Layout 36
4.4. The Commands 37
4.5. Representation of Primitives 38
4.6. Data Structure 39
4.7. Program Flow 42
4.8. Example 44
Chapter 5: The Event Handler Generation Algorithms 47

5.1. Introduction 47

5.2. Description of Event Based Internal Form 48

5.3. The Leading Relation 49

5.3.1. Definition of the Leading Relation 49

5.3.2. Calculating and Checking of the Leading Relation 50

5.3.3. Examples 52

5.3.3.1. Example 1 52

5.3.3.2. Example 2 53

5.4. Error Handling 55

5.4.1. Types of Errors 55

5.4.2. Algorithms 57

5.4.3. Example 58

5.5. Principles of code Generation 59

5.5.1. General Design Principles 59

5.5.2. Generating the C Procedure 61

5.5.2.1. Generating the Tables 61

5.5.2.2. Examples 65

5.5.3. Generating the Event Handler Tables 67

5.5.4. Examples 67

Chapter 6: Implementation of the Code Generator 69

6.1. The Environment 69

6.2. Strengths and Weaknesses 69

6.3. Data Structure 70

6.4. Program Flow 72

Chapter 7: Conclusions 74

7.1. Merits of the RTN Editor 74

7.2. Special Features of the Code Generator 76

7.3. Further Extensions 77

References 79

A1: Example of Designing Dialogue Control 82

1.1. Problem 82

1.2. Recursive Transition Network Solution 82

1.3. Code Generation 84

A2: User Manual 88

2.1. Introduction 90

2.2. Overview 90

2.3. System configuration 91

2.4. Graphical Editor 91

2.4.1. Using the Editor 91

2.4.2. Windows 92

2.4.3. Selection of Commands and Primitives 93

2.4.4. Error Handling 93

2.5. Editor Primitives 93

2.5.1. State 93

2.5.2. State Name 94

2.5.3. Arc 94

2.5.4. Arc Parameter 94
2.5.5. Input Token, Output Token and Application Procedure 95
2.5.6. Network ID 95
2.5.7. Group 95
2.6. Editor Commands 96
2.6.1. Add 96
2.6.2. Delete 97
2.6.3. Move 98
2.6.4. Rename 100
2.6.5. Next Network 100
2.6.6. Add Group 101
2.6.7. Merge Groups 101
2.6.8. List Group 101
2.6.9. Rename Group 102
2.6.10. Destroy Group 102
2.6.11. Help 102
2.7. Associated Facilities 102
2.7.1. Merge Databases 102
2.7.2. Copy Database 103
2.7.3. Destroy Database 103
2.8. Code Generation 103

List of Figures

Figure — Page

1.1 Overview of the Usage of the UIMS 2

1.2 The Logical Model of a UIMS 3

1.3 Summary of the Project Requirements 4

2.1 External Control UIMS Configuration 7

2.2 Internal Control UIMS Configuration 7

2.3 An Example of a State Transition Diagram 9

2.4 An example of an Event Based Model 11

2.5 An Example of Newman's System 13

2.6 Example of Figure 2.5 coded into Network Definition Language 15

2.7 The U of A UIMS Dialogue Control Component Configuration 17

3.1 Example showing the importance of 'wildcard' 28

4.1 The Layout of the Editor's Menus 35

4.2 The Menus Hierarchy 36

4.3 The Window Layout of the Editor 37

4.4 The Data Structure of Group used in the Editor 41

4.5 The Data Structure to Store the RTN in the Database 42

4.6 Initial Layout of the Screen 44

4.7 The Screen Layout After a Diagram is Created 44

5.1 Code Generation Process 47

5.2 Example of EBIF Event Handler 48

5.3 Example Showing the Importance of the Leading Function 50

5.4 Example of Infinite Searching in the Leading Relation 52

5.5 Example Showing How The Leading Relation Files are Formed 52

5.6 Example showing Recursive Calls of the Leading Relations 53

5.7 Tables of Contents of Conversion Program 58

5.8 The State and Ending State Table of Diagram 1 58

5.9 The State and Ending State Table of Diagram 2 58

5.10 Token and Diagram Table 61

5.11 Token and Diagram Table of Diagram1 62

5.12 Token and Diagram Table of Diagram2 62

5.13 The Result of Combining Token and Diagram Tables for Diagram1 63

5.14 Generating Code Algorithm With Arc For Diagram Call 64

5.15 The 'C' Language procedure of the EBIF for Diagram1 66

5.16 The 'C' Language procedure of the EBIF for Diagram2 67

5.17 The Event Handler Table for Diagram 1 68

5.18 The Event Handler Table for Diagram2 68

6.1 Token Record Structure 71

6.2 Token Information Record Structure 71

6.3 Diagram Record Structure 71

6.4 Diagram Information Record Structure 71

A1.1 Main Control RTN 83

A1.2 Two subdiagrams called by 'Main Control' 83

A2.1 Window Layout 92

Chapter 1

Introduction

In recent years, software engineers have realized the benefit of splitting a software system into user interface and application program modules. Many studies have been done on the problem of automatically generating the first component [Buxton83, Kaisk82, Olsen Jr.83b]. This refers to the task of specifying the user interfaces of a system in a high level notation, either textually or graphically, and automatically generating the corresponding user interface module. This generator is an important part of a User Interface Management System (UIMS). There is a strong analogy between a UIMS and a Database Management System since they both provide functions whose detailed implementation are transparent to the user. Moreover, they are both built up of a few separate modules and have a number of notations to describe each module. This thesis describes the construction of a UIMS at the University of Alberta.

1.1. Descriptions of the UIMS

Before UIMSs were developed, software engineers noticed that specification techniques for user interfaces were useful in the development of user interaction modules [Green81, Parnas69]. The specification could be used as a tool for expressing the user interface designers' ideas, for communicating with other designers, and checking for consistency and design errors. In other words, it could provide a general design environment for the user interface module. Later, they realized that many user interface modules (subroutines) were commonly used in many applications. Specification languages were developed to describe the user interface modules and later many user interface generators were built to produce the user interface modules automatically.

Figure 1.1 shows a general overview of a UIMS. Its main purpose is to help the user interface designer to design, implement and maintain the user interface module of

a system. The user interface designer only needs to specify the required user interfaces and the UIMS will generate the code automatically. This simplifies the user interface design since only a high level description of the requirements is necessary and the tedious job of coding can be elimjnated. Detailed descriptions of a general UIMS format can be found in [Green84b, Green84c].

Figure 1.1 Overview of the Usage of the UIMS

There has been a great deal of interest in modeling the UIMS and many different systems have already been built. Nevertheless, most of them follow a common model proposed in the Seeheim Workshop[1] [Green84c]. This model divides the whole system into a presentation component, a dialogue control component and an application interface model (Figure 1.2). The presentation component defines the appearance of an interactive system to the end user. It includes the mapping of logical to physical devices, and deals with the information flows to and from the user. The application interface model describes the set of procedures/modules that can be called from inside the user interface and their required parameters. It forms the interface between the application program and the user interface. The dialogue control component describes the dialogue sequence between the end users and the interactive system and is the main control of the user interface.

The dialogue control component is the most developed component of the UIMS model and can be based on one of three different notations,which are context free

[1] In the Seeheim Workshop, the UIMS is clearly partitioned in presentation component, dialogue control component and application interface model.[Seeheim84]

grammars [Olsen Jr.83b], events [Buxton83] and transition diagrams [Newman68].

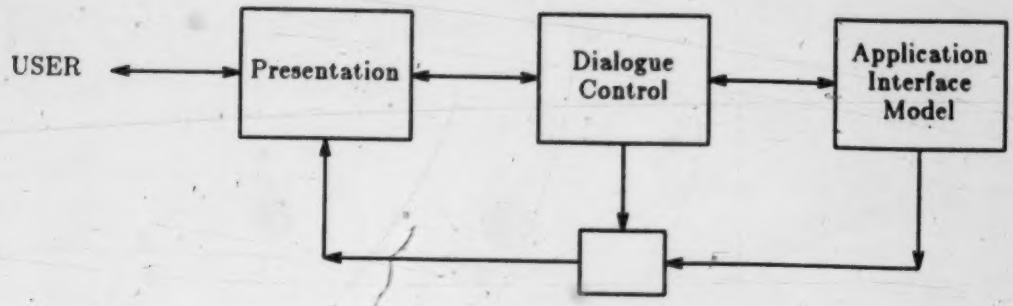

Figure 1.2 The Logical Model of a UIMS

These notations have different descriptive power, but as stated in [Green84e, Green85a], the event-based model[2] has a greater expressive power. Transition diagrams make use of directed graphs in which each node represents a distinct state of the user interaction. A Recursive Transition Network (RTN) is a directed graph that can refer to itself. Two methods can be used for specifying the RTNs, either textual[Rogers81] or graphical. The latter method has the advantage of allowing the user interface designer to enter the specifications interactively and avoids doing the conversion from textual into graphical representation. Also problems are often easier to specify in a graphical way. The graphical approach of specifying the RTNs is used in this project to specify the dialogue sequence.

1.2. Purposes of the project

At the University of Alberta, a User Interface Management System is being built. In this thesis, a component of the system (the dialogue control component) is described. The description is divided into two parts:

(1) A graphical editor for creating and editing the recursive transition networks.

---

[2] The event-based model uses an event based language to describe the user interfaces (i.e. user interactions) of a system. A detailed discussion is given in Chapter 5.

(2) Some conversion algorithms to change the stored RTNs into a special event format.

In other words, a system to support the transition diagram notation of the dialogue control component is built. To provide a system to accommodate all three dialogue control notations, they are all converted into a common format. Since the event-based model has a greater descriptive power, a special event format, "Event Based Internal Form" (EBIF)[3], is designed as the common end.

The system built in this project provides a graphical editor for entering the specification of the dialogue control component and for storing the output in a database. Then a conversion program is used to convert the data into the EBIF. Figure 1.3 summarizes all these requirements.

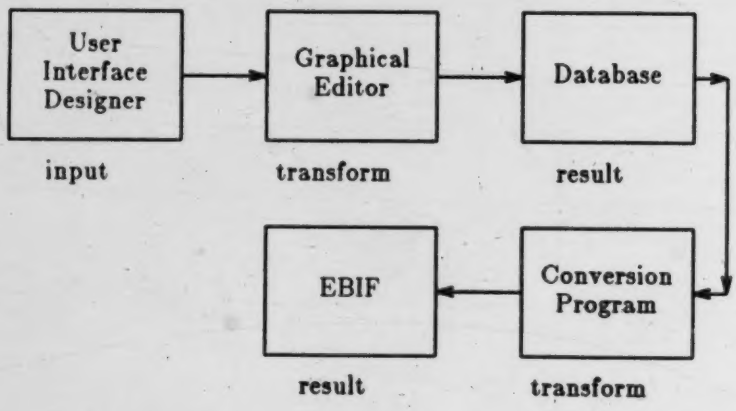

Figure 1.3 Summary of the Project Requirements

[3] The EBIF was designed by M. Green to describe user interactions by means of an event language. A detailed explanation is given in section 5.2.

1.3. Organization of the Thesis

This thesis is divided into two parts. The first part discusses the graphical editor and the second part describes the EBIF code generator. In the first part, background information on the dialogue control component is presented in Chapter 2. In Chapter 3 and 4, the design principles, the implementation of the user interface, the data structures and the operations of the graphical editor are described. Chapter 5 and 6 of the second part discuss the algorithms and coding of the code generating program. In Appendix A1, an example of the use of this system is presented. A user manual for the system (i.e. the graphical editor and the conversion program) is given in Appendix A2.

Chapter 2

Background

In program design, the user interface module is usually separated from the application program. This allows for the automatic generation of the user interface module. An example of using a UIMS to generate the user interface module can be found in [Rogers81]. The UIMS being developed at the University of Alberta is based on the model developed at the Seeheim Workshop (Figure 1.2) [Green84e]. In this model, the dialogue control component is the main control module and many different notations have already been established for it. In this chapter a general description of the dialogue control component and two of its notations, recursive transition networks and the event model, are presented.

2.1. Description of the Dialogue Control Component

In a system that consists of application programs and user interface modules, control can be either external or internal[4]. In the external control configuration, the user interface is in charge of the information flows and it will invoke application modules in response to user inputs. The application programs (procedures) are programmed as discrete functional modules that are available to the user interface when required. On the other hand, the application program is in charge of the control flows when internal control is used. In the internal control configurations, the application calls the user interface modules when it needs to communicate with the user. Figure 2.1 and 2.2 show both structures.

As Rogers said [Rogers81], the external control configuration is well-suited to applications that are designed from scratch or at least heavily rebuilt, since this scheme entails the execution of modules and individual sets of functional components. If the system needs to be changed, these changes are limited to individual modules

[4] There are some systems which don't fall into this classification scheme.

rather than scattered over the whole program. For this reason, most existing UIMSs employ the external control structure. This configuration is assumed in the discussion below [5].

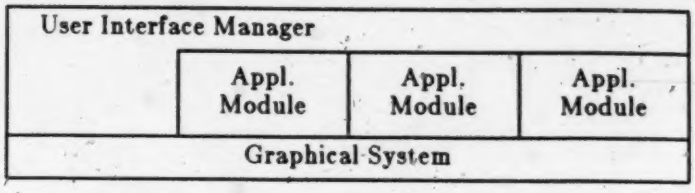

Figure 2.1 External Control UIMS Configuration

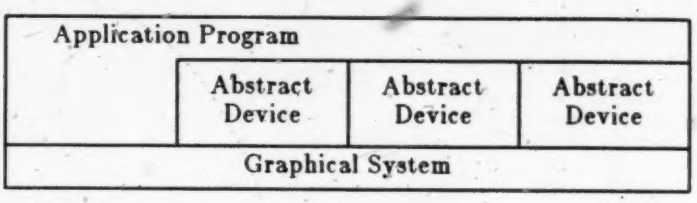

Figure 2.2 Internal Control UIMS Configuration

Two different types of systems have been used for external control configurations, namely glue systems and module builders [Tanner84]. In glue systems all the user interface subroutines are available in libraries and the user interface designer does not need to define the details of these subroutines. The designer only needs to know how to call the subroutines. A special grammar or language is used in the module builders to actually define the interface modules. The designer needs to learn the grammar or the language in order to create his/her own subroutines and library routines are usually not available. The glue system is easy to learn and use but is less flexible due to the limitations of the available programs, which may not meet the user's requirements. The module builder is more general in application but more training is needed by the designer to learn how to write the subroutines. The U of A UIMS uses the module

[5] The University of Alberta UIMS uses the external control configuration since the system is mainly used as a test-bed for user interface design. Nevertheless, the system can also support the internal control configuration.

builder approach to provide a more general design environment.

The dialogue control component defines the syntax of the dialogue sequence. It has the following roles: First, it defines a translation between a time sequence of primitive inputs and a sequence of semantic actions. In other words, it defines the time ordering of inputs and maps a small set of interactive resources into a possibly much larger set of semantic actions. Second, it determines the legal sequences of inputs. Third, it provides special features to handle exceptional cases. For example, when the user enters an illegal input, an error message and a prompt for the user to reenter the data are displayed.

The dialogue control component has a set of legal actions that can be used and a set of object types for manipulation, i.e. to communicate with the application interface module and the presentation component. Also the legal actions are arranged in time order.

In summary, the functions of the dialogue control component are:

(1) Generate the control sequence from the user specifications.

(2) Provide a high level description of displayed objects to make the program more device independent.

2.2. The Recursive Transition Network

The recursive transition network is a convenient way of representing the top level design of an interactive computer system. An early example of this approach is the work of Parnas [Parnas69][6] in which he treated the user as a terminal that was connected to the system. At any instant of time, the terminal was in some specific state that was characterized by the current legal set of inputs and their interpretations. Although the user can type a variety of input messages to the system, all meaningless

[6] Another important system that made use of Recursive Transition Networks was the system proposed by Newman in 1968 [Newman68].

messages are rejected and the remaining commands cause the terminal to switch to new states and some actions are performed during these transitions.

State transition diagrams can be used to represent the above idea. These diagrams are analogous to the state transition diagrams for finite state machines. A state transition diagram is a directed graph which consists of nodes (states) and branches (arcs). States represent polling loops or dynamic stops in the program where they wait for actions from the user. The arc is a directed curve that connects a starting state to an ending state (i.e. from a tail to a head state). Associated with each arc is an input message that must be matched with user's input in order for the system to traverse that particular arc. Also included are an optional output message and a procedure name[7]. The former contains feedback information or prompts for the user, and the latter procedure will be executed if the corresponding arc is traversed. Figure 2.3 shows a logon example for an external control UIMS configuration (simplified from [Parnas69]) where the user can logon to a computer system, inquire information about the load or read the news before logon.

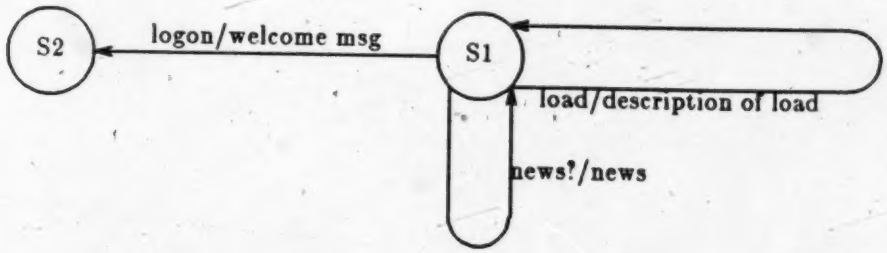

Figure 2.3 An Example of a State Transition Diagram

In this example, the initial state is "S1". In this state, the user can type "news?" to read the news before logon. The system will generate the command to display the news and return to state "S1". If the user types "load", the workload of the system is

[7] The arc parameters are arranged in the format "input message/output message/procedure name".

shown. Once the user types "logon", the system switches to state "S2" and prompts with a welcome message. Further actions can start from "S2", although it is not shown in Figure 2.3.

Two methods can be used to represent the state transition diagrams, either graphically (as shown in the example) or textually. In the graphical representation, states, arcs, input messages, output messages and application procedures names are used together to represent the transitions of the dialogue(i.e. to control the dialogue sequence). The user only needs to specify these parameters, as in the example, through the use of a graphical editor (e.g. [Newman68]). In the textual format, the graphs are converted into textual statements before entering them into the system. Free format data files are often used for this purpose.

Because of limited screen area, the whole dialogue sequence cannot fit in a single screen. So the RTNs are usually partitioned into a number of levels. In most existing systems, the textual representation has a number associated with each statement indicating the level at which the statement resides. For graphical representations, a number of state transition diagrams are created and linked together to form a hierarchy.

The advantage of using a graphical representation is its ease of implementation as no conversion is required and direct interaction is possible.

2.3. The Event Model

In the event model, the interactive dialogue is seen by the system as sequence of events, each of them represents either an input from the user or an output from the application program that influences the flow of the dialogue. Each event is associated with several event handlers. When an event is generated, it is sent to one or more event handlers that have included the event in their event lists. In each event handler, different sets of statements will be executed depending on the generated event. These

procedures may be used to update the screen, create another event handler, destroy existing event handlers, call application programs, do calculations and so on.

The event model is more procedure-like and has a greater expressive power than the other notations since it is equivalent to Turing machines, while the other two notations have the power of push-down automata [Green85b]. The event handler for the logon example above is shown in Figure 2.4. Since the U of A UIMS is implemented on a VAX[8] running the UNIX[9] operation system, the event language is similar to the "C" programming language. This saves the designer time in learning and understanding a new language.

```
Eventhandler logon is
 Token
  keyboardstring1 logon;
  keyboardstring2 news?;
  keyboardstring3 load;
 Event logon {
  print_mess( welcome_msg );
  state = S2;
 }
 Event news? {
  print_news;
 }
 Event load {
  process_load();
 }
End login;
```

Figure 2.4 An example of an Event Based Model

The first line of the event handler is its name "logon". Following are three strings that can be entered from the keyboard and their equivalent event names. The rest of the event handler are the names of the events and the actions that they will performed when the events are generated. For example, when "keyboardstring1" is entered from the keyboard, an event called "logon" is generated. This event calls the procedure "print_mess" to print the welcome message and changes the system state to "S2".

[8] VAX is a trademark of Digital Equipment Corporation.
[9] UNIX is a trademark of AT & T Bell Laboratories, Inc.

2.4. Review of Previous Work

During the past decade, many systems have been built which automate the generation of the user interface module from a high level specification. Most of these systems are based on either context free grammars or recursive transition networks. Examples of the first notation are "SYNGRAPH" [Olsen Jr.83a], "FLAIR" [Wong82] and "ICAN ICUE" [Waervagen83], while the RTNs based systems include "SYNICS" [Edmonds81] and Newman's system based on the Network Definition Language [Newman68]. Although the event model is another major notation, it is not well developed and few systems have been built around it. Since this project is based on RTNs, an example using Newman's system[10], is described below to give some insight into what has been done in the past and what is required now.

2.4.1. Newman's System

The system that was introduced by Newman for specifying user interfaces more than ten years ago uses a graphical way of describing dialogue sequences. Besides having states and arcs as other systems do, it includes features such as "program block", "instruction for execution", "test routine" and "response" to enhance the specification power. Normally, when the user input matches the action defined in the arc, branching will take place. However, this can be over-ridden by making use of the test routine included in the branch definition. Branching can also be started by system actions(i.e. the result of a procedure may determine which of several states the program will branch to). This type of procedure is called a program block. The user interface designer can also specify the response of a system whenever a particular state is entered. This is done by making use of a "response" facility attached to that particular state. A message is included in the response that will be displayed when the state

[10] The RTN system presented in this thesis is based on the idea of Newman's system. This is the reason why it is described in detail.

is entered. The reaction of the system when a particular arc is traversed is represented by an "instruction for execution". An example of these features is shown in Figure 2.5[11].

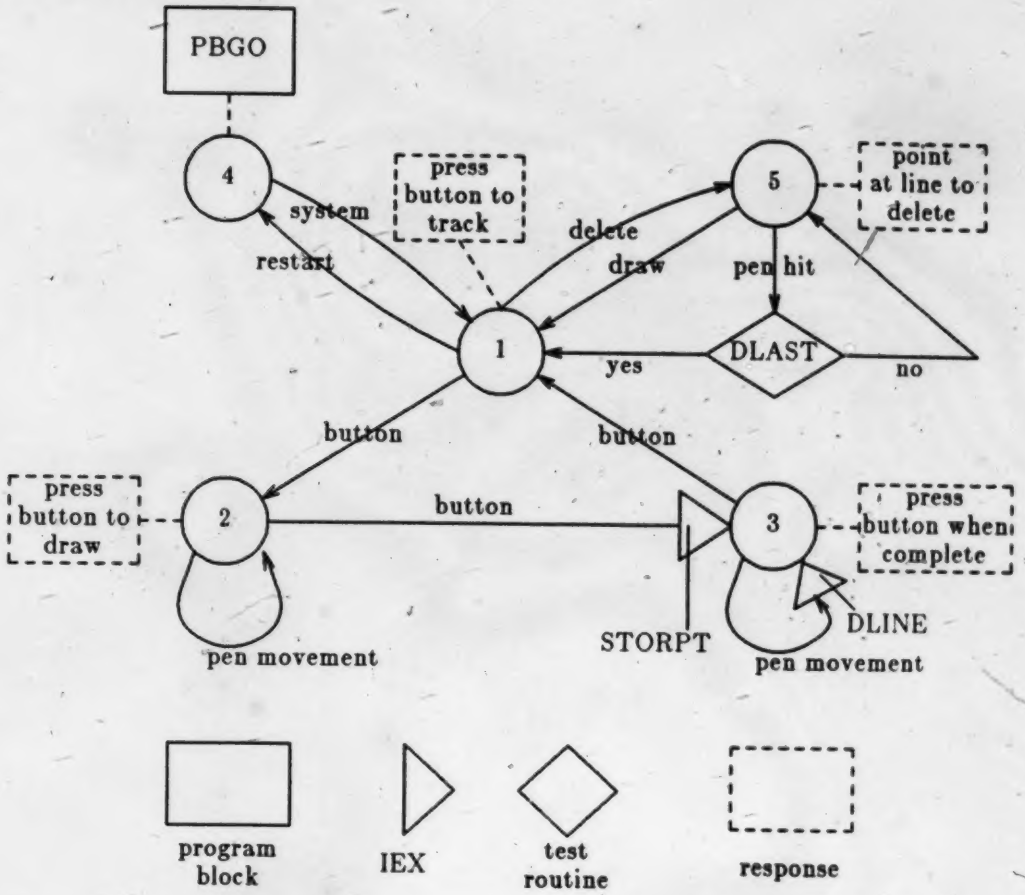

Figure 2.5 An Example of Newman's System

In this example, assuming the initial state is "1", the user can enter the commands "button", "restart" or "delete" to change the system state to "2", "4" or "5" respectively. If "response" is associated with a state, the particular response is displayed when the state is reached. For example, when the user enters the command

[11] Extracted from the paper called "A System for Interactive Graphical Programming" [Newman68].

"button" to change the system state to "2", the response "press button to draw" is shown on the screen. When the system switches from state 2 to state 3, the instruction "STORPT" is executed. A program block "PBGO" is associated with state 4, so when the system switches to this state, the procedure in "PBGO" is executed.

At the time when Newman's system was developed, computers were not very common and most jobs were running in a batch mode with data entered off-line. The state transition diagrams were usually prepared off-line so a special language called "Network Definition Language" was developed to describe the networks. Each state was given a number and the branching instruction was based on this numbering system. Actually this language is just a direct translation of the diagrams. Figure 2.6 shows the equivalent Network Definition Language for the above dialogue. The left hand column shows the commands/inputs and the right hand column contains comments. The first line of each block of statements is the name of the state. "RESP", "IEX", "PB" and "TEST" represent the "response", the "instruction for execution", the "program block" and the "test routine" respectively.

INPUT		COMMENT
STAT	1	State definition. State 1
RESP	PRESS BUTTON TO TRACK	State 1 response. "Press button to track"
ACT	0	Branch definition, action of category 0 (command)
MES	RESTART	Message "restart"
SE	4	State entry, i.e. branch leads to state 4
ACT	0	Branch definition:command "delete" leads to state 5
MES	DELETE	
SE	5	
ACT	10	Branch definition, category 10 (button)
SE	2	Pressing button leads to state 2
STAT	2	State 2 definition
RESP	PRESS BUTTON TO DRAW	State 2 response
ACT	7	Branch definition, category 7 (pen movement)
ACT	10	Branch definition: pressing button leads to state 3
IEX	STORPT	STORPT stores pen position as starting point when button is pressed
SE	3	
STAT	3	State 3 definition
RESP	PRESS BUTTON WHEN COMPLETE	
ACT	10	Branch definition: press button leads to state 1
SE	1	
ACT	7	Branch definition, pen movement
IEX	DLINE	Dline computes and displays fresh line at every pen movement
STAT	4	State 4 definition
INIT		Initial state, program starts here
PB	PBGO	Program block PBGO executed on entering state 4
ACT	5	Branch definition, category 5 (system)
SE	1	Completion of PBGO leads to state 1
STAT	5	State 5 definition
RESP	POINT AT LINE TO DELETE	
ACT	0	Branch definition: command "draw" leads to state 1
MES	DRAW	
SE	1	
ACT	6	Branch definition: category 6 (pen hit)
TEST	DLAST	Test routine DLAST deletes indicated line
SE	1	If last line, branch to state 1
END		

Figure 2.6 Example of Figure 2.5 coded into Network Definition Language

2.5. Summary of Existing Systems features

The example above shows a typical RTN system and it is easy to see that textual descriptions usually make use of a numbering system to define the nodes. This method is efficient if the diagram is complicated and the number of hierarchical levels is large, however there is no immediate system feedback. The user cannot make sure that the entered diagram is the same as expected. What remains to be done is to have a graphical editor to enter the state transition diagrams interactively for easy debugging and modification.

2.6. The University of Alberta UIMS Requirements

In the University of Alberta UIMS, all three notations, recursive transition networks, context free grammars and the event language, are supported for describing the dialogue control component. The user can choose his favourite method for specifying the dialogue. In order to allow this, a special event based language, "Event Based Internal Form" (EBIF), was designed that serves as a common format and all three notations are converted into it. Figure 2.7 shows the configuration that is used in the University of Alberta UIMS. This thesis is using RTNs to describe the dialogue sequence so two steps are needed. First, a graphical editor is used to create the transition diagrams and store the RTNs in a database. Second, the data is converted from the database into the EBIF. Detailed descriptions of the EBIF and the dialogue control component requirements can be found in [Green84e].

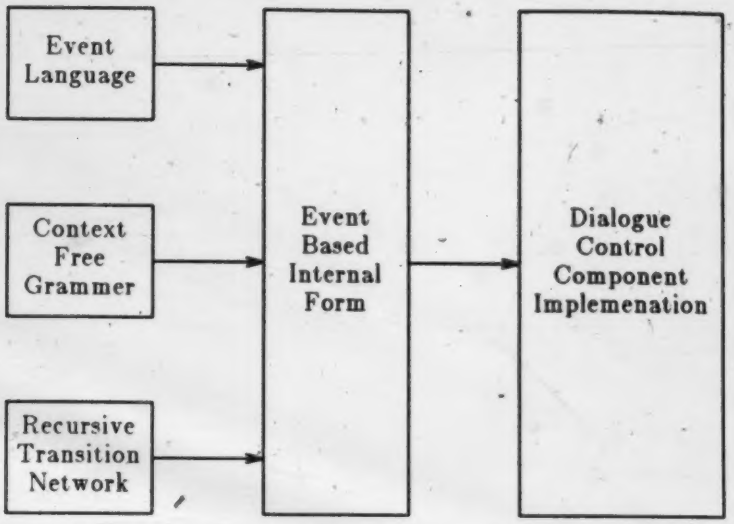

Figure 2.7 The U of A UIMS Dialogue Control Component Configuration

Chapter 3

Designing the Graphical Editor

3.1. Introduction

An interactive approach is used for specifying the recursive transition networks in the University of Alberta UIMS. A graphical editor has been designed and implemented in this project in order to provide a design environment for the user interface designer to do this job. The editor is divided into user interface module, commands and primitives modules, and modules to update the database and screen. This chapter describes the basic aims and the design principles of the editor as well as how the various modules are designed. In the next chapter, a detailed description of the implementation is given. Basically, the RTNs have a similar format to Newman's state diagrams as shown in Chapter 2, Figure 2.5.

3.2. Basic Aims

The graphical editor is designed to create and edit the recursive transition networks interactively. It obtains the input data from a keyboard and a tablet/mouse and the output is sent to a database for storing the created diagrams. A monitor is used to show the system's feedback and the transition diagrams. Hardcopies of the edited diagrams can also be produced. The basic requirements of the editor are the provision of commands to enter and modify the networks. The editor is used interactive so user friendliness is an important requirement and a large portion of the design and implementation is devoted to providing such a environment. The editor is designed to be used by both novices and experts.

The design of the editor commands and primitives (objects) is a difficult task. The commands and objects should cover all the general editing requirements of the RTNs; however, only minimal overlapping should exist between them. That is, each

command performs a distinct function. The other main consideration is the design of an efficient data structure to provide a short access time for interactive use. The menu layout, the functions of the keys on keyboard, the use of the buttons on the tablet/mouse, and the editor's response to the user's inputs are important as well because they help the user to develop a model of the editor. Lastly, the common errors found in the editor and the editor's error handling algorithm are discussed. In addition to all these requirements for the editor, other supporting facilities for the manipulation of the RTN database are also discussed in this chapter. As a summary, the basic design of the editor includes:

(1) Designing the user environment.

(2) Designing the editor's commands and objects (primitives).

(3) Designing the data structures.

(4) Designing the menu layout.

(5) Discussion of the error handler and the error types.

(6) Designing the supporting facilities.

Each of these points is discussed separately in the following sections.

3.3. Designing the User Interface

When the user interface module of the editor was designed, emphasis was put on the following points:

(1) Both novices and experts can use the system, i.e. they are both potential users.

(2) The editor users should not need to remember a lot of commands.

(3) Immediate feedback to the user's inputs.

(4) Development of a user model so the designer can learn the editor commands easily.

Since the user interface is the most important component in the graphical editor, it is the first module to be designed and each of the above points is discussed in the following sections.

3.3.1. Potential Users

Both novices and experts are potential users of the editor and they are treated differently by the system. For the first type of user, more guidance and prompting is required to teach them how to interact with the editor. Prompting messages is one approach in human-computer interaction. Whenever the editor expects the user to enter data or a command, a prompting message is displayed. In this case, the novice users can depend on these messages and need not refer to the manual for information on the operation of the editor.

The trade-off in the above method is a decrease of execution speed due to displaying the long prompting messages every time the user interacts with the editor. The expert users have models of the editor in their mind and know the required sequence of interactions. For them, this decrease of execution speed is undesirable and the messages can be ignored. They want the system to operate as fast as possible. The editor uses a flag which can be set by the user to specify whether prompting messages are required. In this way, the system can fit both types of users.

The editor provides the basic commands which are sufficient for editing any RTN. Some advanced features are also available. They perform the same functions as the basic commands, but in a more efficient way. However these commands are more complicated to use and a longer learning time is required. The novice users are unaware of their existence and are restricted to the basic commands, which are enough for normal usage with lower operation speed. In this way, the editor is working at a basic level for novices, but at an advanced level for experts. Having the advanced features overlapping with the basic commands, the novices do not need to spend a long time

learning the editor commands and their syntax before they can start creating RTNs.

3.3.2. Minimal Memory Required

In traditional editors, the end users need to remember all the commonly used editing commands and their syntax. For a complicated editor, a long learning time is required. The graphical editor makes use of the stationary-menu/moving-cursor approach to reduce the burden on the user's memory. The commands as well as the manipulated objects are all shown on menus so the end user only needs to remember the contents of the various menus, but not the syntax of the commands for execution. Both stationary and pop-up menus were considered for commands and object selection, but the latter method was discarded due to the long redraw time whenever menus are erased.

Another facility provided by the editor to reduce the burden on the user memory is a "help" command. Every command and object on the menu can be called for "help" to give an detailed explanation of its action and meaning. So whenever there is some confusion on the use of commands and objects, either in their meaning or their syntax, or when the editor reports an error that is not understood, the user interface designer can refer to these "help" messages. The on-line "help" system can be used while the diagrams are being entered or edited.

By making use of "help" and menus, the novice editor users can start to create the RTN almost without any knowledge of the editor's commands.

3.3.3. Immediate Feedback

In user interface design, an important aspect is to let the users see what they have entered, and have immediate feedback from their actions. The graphical editor responds to the user whenever something is entered, either from the tablet or the keyboard. This feedback is especially important for computer systems under heavy workload. It notifies the user that the editor has accepted the command and is processing it. If no indication is shown, the user may suspect the command entered is being ignored by the editor, and may reenter the command again. This is undesirable because the command is executed twice.

Users are the most unpredictable component of the editor and they can enter any command into the editor, both valid and invalid. For legal inputs, the action is shown as a response on the screen. If the command is a complete operation or the last command to complete a sequence of operations, the result is drawn on screen, e.g. a command to draw a state will display a circle on screen, or when the last point of a Bezier curve is entered, the curve is shown. For an action which needs several commands for completeness, intermediate acknowledgements are shown before the final result is displayed. In the editor, four points are required to draw an arc and when the first three points are entered, they are positioned as crosses on the screen. For an illegal input, an error message is displayed explaining what the editor is expecting and requesting the user to reenter the command. In this way, the user can have confidence that the input commands are in the correct format and need not worry that the editor will complain after a long sequence of commands has been entered.

Error detection is performed interactively with the user's input. The main reason for this is to provide an immediate chance to correct the errors or to call for more explanation. Detecting and fixing the bugs instantaneously provides a better design environment because the user knows exactly where the bugs are and can fix them

easily. If an error is detected later by the assembler or the code generator, reloading of the diagram from the database is required.

Highlighting the command and primitive (object) the user has selected is used to remind him of his selections. In some systems, this can be done by highlighting the selected options on the menus directly. However, multiple menus are used in this editor, but only one menu is displayed at any time. So menus need to be redrawn and erased very often. Since options are chosen from different menus this method cannot be used in this graphical editor. The other method, which is used in this editor, is to create a separate window to display the command and primitive options as well as the diagram name that is in use. Whenever the user selects a new command or primitive, this window is updated immediately so the designer can be sure of the command he is using.

3.3.4. User Model

A powerful idea in user interface design is to help the user create a model of the system himself/herself. Once this model is formed, the user can use the editor in his own way based on the model. This development can only be done when the user understands what the editor is doing, the input sequences it is expecting, and the usage of the function keys. When the novice starts to use a new system, a model of the editor is formed in which the functions of the keys and buttons as well as the use of commands and primitives are included. By designing the commands to perform unique functions, and by the assignment of function keys to distinct functions, a clear model of how the editor is operating is easy to develop. This can greatly enhance the development of the user model.

The tablet/mouse has a lot of buttons that could be used by the system, but the editor only uses two of them. By minimizing the number of keys used and the assignment of unique functions to them, the user can easily remember the functions of the

keys. If the keys are used for different functions in different situations, the user needs to remember a lot of combinations and can easily be confused.

3.4. Designing the Editor Commands

3.4.1. Basic Commands

The requirements of the basic editor commands are non-overlapping and uniformity. Non-overlapping means every command performs a distinct function and no two commands can do the same thing. Uniformity means that commands are used in the same way and have the same meanings under all circumstances and applications. For example, "rename" can be used to rename a state or a diagram. The last basic requirement is that the command set be complete, i.e. all basic editing functions can be performed by a unique command and a sequence of commands is not required to achieve a single goal.

The basic commands are addition, deletion, removing and renaming of objects (primitives). Their meanings are self explanatory. They cover all the commonly used editing functions that the user expects. The commands can apply to all types of objects except some meaningless combinations, for example, adding an arc parameter to a diagram before the arc is created.

For all operations which will destroy a great deal of information that cannot be restored easily, the editor will ask for confirmation before it is executed. For example, the command to remove a diagram (i.e. to delete the diagram from the database) will ask for confirmation before actually executing. This double checking facility is especially useful for novice users.

3.4.2. Auxiliary Commands

Some special commands are also available, including "Next Network" and "Exit". "Exit" allows the user to quit from the editor. Two options are available, either quit the editor with the current session's modifications saved or not. This is useful for users if they change their mind later and want to restore an old version without saving the current session's work.

Another command is "Next network" which is used to edit a new diagram. If the diagram exists in the database already, the diagram is loaded; otherwise a new entry is created in the database. All the diagrams edited in the same session are stored in one database with the name of the root diagram as the database anme, i.e. the name of the database is the diagram name passed to the graphical editor as an argument when the editor is called.

"Grouping" is a set of advanced commands designed for expert users. Their functions can be accomplished by the 4 basic commands as stated above, but with less efficiency. This set of commands collect objects together to form groups and then the other editor commands can use these groups in the same way as the individual objects. This is an advanced feature which reduces the number of commands the user must use. For example, when the user wants to copy a group of objects from one diagram to another diagram, this can be done very easily with the "grouping" command. First, the objects to be copied are grouped together. Then when the "inserted" diagram is loaded into the editor, an "add" command can be used to add the group of objects in a single command. Another feature of this grouping command is that once a group is defined, it is remembered for later use, even in another session. So patterns which are used very often in the RTNs can be stored as groups in the database and added to the desired diagrams whenever necessary. This reduces the time to create them every time they are used.

3.5. Design of Manipulated Objects

3.5.1. Basic Objects

There are four basic objects used by the editor. States and arcs are the two most obvious ones. They are represented by circles and curves with arrows. State name is another basic object. All states have names associated with them and these names can be added, deleted, moved or renamed in the same way as the states, so they are treated as another primitive. The other main object is the arc parameter which specifies the input tokens and the output tokens to be processed as well as the application procedure tokens to be called when the arc is traversed. In existing RTN systems, the arc parameters are associated either with the arc itself or the head state of the arc. In this design, the first alternative is chosen because it is easier to implement and also the diagram display is less cluttered. If parameters are linked with the states, then two pieces of information (the arc parameter and the state name) need to be shown for each state and the diagram layout is more confused.

Input tokens are the tokens sent from the presentation component to activate the transition of states. When the user enters an input from either the tablet/mouse or the keyboard, the presentation component will send a corresponding token to the dialogue control component. The output token is used to send information to the presentation component, for example, the dialogue control component can send a token to the presentation component to update the screen. If the user wants to call a procedure in the application program, an "application procedure token" is sent to the application interface model to do this job.

Besides these four basic objects, the input token, the output token and the application procedure token can be manipulated individually. The main advantage this feature is allowing the user to change individual tokens without re-entering all the arc information. The editor does not allow the user to create one of the components of the

arc parameter by using the "ADD" command. The same function can be achieved by using the basic object "arc parameter" with the two unused token fields left blank. This is a result of the non-overlapping nature of commands.

3.5.2. Auxiliary Objects

Transition diagrams are the next object that needs to be manipulated. All meaningful commands can be applied to them of which "delete" and "rename" are the two most useful ones. A separate set of commands could be designed for the diagrams which treats the diagrams as a different type of primitive. However, for the consistency of the editor's commands, the same command set is used. This reduces the burden on the user's memory because only one set of commands needs to be remembered.

As an example of the uniformity of editor commands, "ADD" is one of the editor commands that is frequently used. All primitives except "network"[12] can use this command. The exception is due to the fact that "add" means inserting an object into a diagram. When a "network" is added, the editor can either load an existing diagram and display it on the screen or create (i.e.insert) a new entry in the database. That is the reason why another command "Next network" is designed to do this job because "ADD" does not include the loading of primitives from the database.

A "wildcard" token name can be used as the input token field of an arc parameter. If the event handler finds no matching input token in the current state, the statements corresponding to the "wildcard" token are executed. This token is very useful in many situations, for example if the designer wants the user interface to pass program control to an error handling subroutine when there is no matching input token in the current state. The designer only needs to add another arc with the "wildcard" as the input token field and use the special error handling subroutine as the application procedure

[12] "Network" is the name used in the editor to represent a transition diagram.

token. Figure 3.1 shows such an situation in which "token3" is entered by the user. If the arc with the wildeard did not exist, the system error handler would be used. However, since the wildcard can match any token which is not found on the other arcs with the same tail, the special error_handling subroutine "err_handler" will be used.

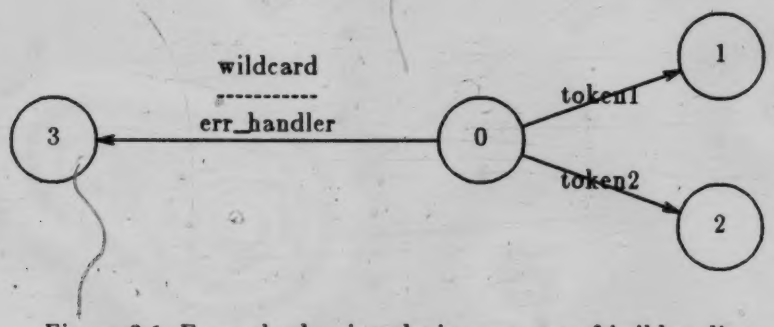

Figure 3.1 Example showing the importance of 'wildcard'

The last object to be designed is "group". Groups are treated in exactly the same way as the other objects. Objects can be collected by specifying the lower left and upper right corners of a rectangle or by selecting objects one by one to form a group. Both methods are useful, for example if the user wants to move a section of the diagram's objects to another diagram, the first grouping method is used. When selected objects are moved, the second choice is used. After the group is formed using the above methods, it is assigned a name and this name can be used in exactly the same way as the other primitives.

Three decisions have been made in grouping elements:

(1) Whenever a state is selected, the state name is selected automatically.

(2) When the arc is selected, the arc parameter is selected as well.

(3) If objects are selected by specifying the corners, all the objects that are not completely inside the rectangle are ignored.

3.6. The Data Structure

One of the important activities in the design phase is to design a data structure which is easy to manipulate and fast to access especially for real time applications. The four basic primitives, states, state names, arcs and arc parameters, are the essential parameters of a diagram which must be saved. They should be stored in a way that retrieval and modification can be done quickly and related information is linked together. Since the state and the state name are closely related, they are grouped in the same record. The arc and the arc parameters can be put in a record in a similar way as the state and the state name, but this method is not used in the editor. The main reason is that both records are big and in the graphical editor some operations only need to access one type of record. If they are grouped together, almost twice the amount of data needs to be accessed before the selected record is located. The system overhead in paging and swapping for accessing data will slow down the operation. So the editor uses two separate record types for them.

Pointers are used in the editor's data structure to link all related data together. A linked list of diagram records is created to form a network of diagrams. Within each diagram, there are linked lists for the states (and the state names), the arcs, and the arc parameters in order to make it possible to access individual fields directly without going through a lot of searching. No record contains duplicate information, this ensures consistency of data and all updating can be done in one search.

Another main data structure is used to store the groups and their elements. Each type of element within the same group is collected together. So three linked lists are used, one to link all the groups together, the second linked list is used to link all the header records of the elements and the last one is for grouping all elements of the same type.

3.7. Menu Layout

Both pop-up and stationary menus could have been used as the menu type. The first method is not used because the time to redraw the screen when the menu is erased is quite long especially when the computer system is working under heavy load. For stationary-menu/moving-cursor menu, it is not possible to put all available options on one menu so the selections are grouped into several menus. A hierarchy of menus (i.e. a tree structure of menus) is a possible arrangement, but it has a serious disadvantage that whenever the user wants to switch from one menu to another, he needs to go back to the parent menu before a new menu can be chosen. Sometimes, he may need to go several levels upward before he reaches the desired menu. This is not a good environment for selection. The other alternative is to construct menus at the same level in which each menu can directly call the other menus. This can greatly improve the operation speed but occupies more menu screen space since all the other menus' names are on each menu. This editor uses a combination of the two methods in which a two level menu arrangement is used and enjoys the advantages of both methods. This menu structure contains a number of branches but all have a tree height of two. In the worst case, the user only need to go up one level before menus in other branch can be selected, also only the parent menus' names are on the other menus.

The options (i.e. commands and primitives) are grouped into a few menus types with each type doing a special function. For example, one of the menus types is used solely for commands and the other is used only for primitives. Within each menu type, i.e. within each branch, a few sub-menus may be required for holding all the options and one of them is selected to be the master.

3.8. Error Handler

The graphical editor is an interactive program and the user is prone to make errors so the error handler is a very important module in the design. It is designed to detect all errors that are caused by the user and report to the user immediately. Errors can be classified into two types and they are discussed in the following two sections.

3.8.1. Operation Errors

The first type of error are the errors in using the editor. These errors are mainly caused by the user who is not very familiar with the operation of the editor or the sequence of entering data. Possible causes of these errors are:

(1) Press wrong key in operations.

(2) Incorrect sequence, for example assign a state name to a state before it is created.

(3) Illegal input format, for example use letter instead of number for state name.

(4) Meaningless operation such as move a diagram.

All these errors are detected by the editor and an error message is printed saying what is wrong with the command and asks the user to reenter the command or choose another operation.

3.8.2. Inconsistent Errors

The other main error type is errors which cause the diagram to go into some inconsistent state. The possible errors are:

(1) The arc parameters do not include an input token field. In this case, there is no way to invoke this arc.

(2) Assigning two names to the same state.

(3) Assigning two arc parameters to the same arc.

This is a first level of checking for syntax errors. The final checking is done at code

generation time.

A special button on the tablet is available for the user to undo an incomplete operation. Once this key is pressed, the current operation will stop and the editor restores to the state just before the operation. This is an important function provided by the editor to restore it to some known state when the user finds errors in operation, or he wants to change his mind and restore to the previous state.

3.9. Supporting Facilities

A few support tools are available to manipulate the resulting database. They treat the database as an ordinary datafile. The basic facilities provided are:

(1) A program to destroy an existing database. The editor can do the same job but all the diagrams in the database are destroyed one by one. This program will destroy all the diagrams stored in the same database at the same time.

(2) A program to merge two databases together.

(3) A program to copy one database to another.

Chapter 4

Implementation of the Graphical Editor

4.1. Introduction

The implementation of the graphical editor is divided into three separate parts. They are responsible for separate functions and can be classified as:

(1) user interface.

(2) editor's commands, the system responses and the error handler.

(3) updating of the data structure and the screen.

Each of these modules is discussed in the following sections.

4.2. The Environment

The graphical editor is written in the "C" language and runs on a VAX[13] 11/780 under the UNIX[14] operating system. The editor needs a graphics terminal for display, a tablet/mouse with at least two buttons for input and an ASCII keyboard. Two supporting packages are required by the editor. A graphics package "WINDLIB" is used for handling all the graphical interactions and displays and a database called "FDB" is used to store all the data.

Windlib is an event driven window based graphics package in which a number of windows can be created with their locations and sizes fixed by the user. The windows are arranged in an hierarchical structure and an event handler is associated with each window. Each event handler handles the interactions with the user and controls all the operations within its window. The implementation of the editor is based on these event handlers.

---

[13] VAX is a trademark of Digital Equipment Corporation

[14] UNIX is a trademark of AT & T Bell Laboratories, Inc.

FDB is a Frame DataBase which is similar in structure to a pointer type of data structure. The database is made up of frames and each frame can hold various numbers of data items and pointers called slots. This structure is basically the same as the "record" type in Pascal. The data structure that has been designed for the editor can be used directly with FDB without modification. Detailed descriptions of Windlib and FDB can be found in [Green84a, Green84d].

4.3. Implementation of the User Interface

The most important considerations in the implementation of the user interface are the use of the screen and the layout of the various windows. This determines the external appearance of the editor. The layout of the menus and how to call one menu from another is the second main consideration. This includes the methods of choosing commands and primitives from menus.

4.3.1. The Menus

The editor collects commands and primitives with similar usage and function into the same group. A master menu in each group contains the most commonly used alternatives. Several sub-menus may be required to hold the remaining options of the group. Whenever the user calls another group, the master menu is the one called and it is the first menu to be invoked within that group. This arrangement of menus has two advantages:

(1) A two level menu structure is formed in which callings within the same type or between different types is easy.

(2) It decreases the screen space occupied by the menu names on each menu. If each menu name is on all the other menus in order to call it directly, a large portion of the area is used solely for this purpose. More menus are required to hold all the available options because of this waste of screen space.

Basically four types of menus exist with two important ones, which are the command and the object menus. In addition to these two types, there is a menu to exit from the editor and a menu for the "grouping" commands. The layout of the menus is shown in Figure 4.1 with Figure 4.2 showing the calling hierarchy of the menus.

COMMAND MENU	OTHER COMMAND MENU	EXIT MENU
Return to Object Menu	Return to Main Menu	Return to Main Menu
Add	Next Network	Quit and Save
Move	Dump Database	Quit without Modification
Delete	Grouping	Help
Rename	Help	
Other Command	Exit	
Help		
Exit		

OBJECT MENU	OTHER OBJECT MENU	GROUP MENU
Return to Main Menu	Return to Object Menu	Return to Main Menu
State	Group	Add Group by Location
State Name	Input Token	Add Group by Pointing
Arc	Output Token	Merge Group
Arc Parameters	Application Procedure Name	List Group
Other Objects	Network ID	Rename Group
Help	Help	Destroy Group
Exit	Exit	Help

Figure 4.1 The Layout of the Editor's Menus

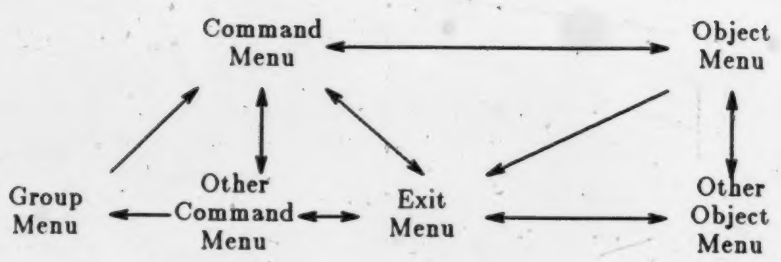

Figure 4.2 The Menus Hierarchy

Selection in the menu is done by pointing at the menu item and pressing a button on the tablet. The user can choose another option by selecting from the menu again. The chosen command and primitive are shown on a separate small window to remind the user what has been selected. Whenever a new option is chosen, this window is updated immediately.

Another facility provided by the editor is the availability of a "help" command on every menu. The user can call for help to get an explanation of all the options on the current menu. This seems to be a limitation of the editor since only the information for the options on the current menu are available for help. However, this limitation is intended because "help" is designed to be an on-line facility so it is more likely to be useful on the current menu.

4.3.2. Window Layout

Due to the limited screen space, not all information can be displayed on the screen. As a result only one of the six available menus is shown on the screen and a window is used to display it. Basically three other pieces of information and hence three more windows are required. First, a work area is used to show the current RTN that is being edited. Second, an error/prompt message window is used for displaying the error messages or prompting the user for input. The third window shows the

selected command and primitive as well as the current diagram name. The layout of the various windows on the screen is shown in Figure 4.3.

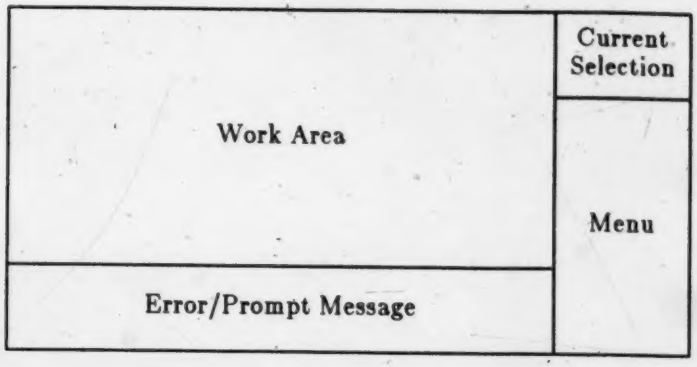

Figure 4.3 The Window Layout of the Editor

4.4. The Commands

The basic commands are addition, deletion, removing and renaming. They can be used for all meaningful primitives, including states, state names, arcs, arc parameters, diagrams and groups. Whenever the user presses the select button in the work area (in this case, select a position), the command is executed and the same command is held until the user selects another option. The user can choose primitives and commands in any order. The currently selected commands and objects can be changed by selecting again[15]. So the editor command can be said to have no syntax at all.

There are some special commands which are useful. The first one is "grouping" which is used to collect objects together to form a group and assign a name to it. A database is created to store all the group elements and their names so they can be used in a later session. This database uses the same name as the RTN diagram with ".g" appended at the end. Whenever the user trys to add a new group to the database, the existing groups names are first checked to ensure no duplicate names are used. The

[15] Only the commands or objects to be changed are selected again.

two methods as stated in the last chapter (section 3.5.2) are used to collect objects.

Another command is "exit" which is used to exit the editor with or without the current session's modifications saved. When the editor is called, a new database can be created by calling the graphical editor with a new diagram name as its parameter. If the user changes his mind and quits the editor without saving the modifications, this database is destroyed. Otherwise, the database having the same name as the root diagram is saved.

"Next Network" is another special command to jump to or load another diagram. When this option is chosen, the editor will prompt the user to enter the next diagram's name. If the name exists in the database already, the diagram is loaded and displayed on screen; otherwise a new diagram is formed and a new database entry is linked with the existing diagrams. The screen is cleared to indicate that a new diagram has been formed.

4.5. Representation of Primitives

Four basic primitives are used in the editor and they are represented in unique ways within the editor. Circles are used to represent states. For arcs, the easiest method is to use straight lines going between the centers of the states with an arrow head at one end. However, it cannot handle the cases of more than one arc going between two states because the two arcs will overlap. Sometimes the user may want the arc to be a curve in order to go around some states in between the two end points. Two approaches can be used to specify the locus of the arc. The first method is to let the user draw the arc on the screen and the editor traces and samples all these points. The other method is to use a Bezier curve in which only 4 points including two points on the two ending states are used to specify the locus. The latter method is used in the editor due to the smaller amount of memory required to store all the points, although the locus may not be exactly as expected.

Before the state names and the arc parameters are entered into the diagrams from the keyboard, their positions need to be specified. State names are associated with states so their positions are specified by just placing the tracking cross inside the corresponding circles before the names are entered. The state names will be centered inside the circles automatically. For the arc parameters, they can be placed at any point on the diagram. The editor can assume the arc with the minimum distance from the parameter is the associated arc. The user just needs to place the tracking cross on the intended position and enter the parameter. However, this method has two drawbacks. First, finding the minimum distance is not trivial because only four points are stored for each arc. Second, the arc with the minimum distance may not be the desired arc. So the graphical editor requires the user to specify the two end states for the arc by using the tracking cross. Another advantage of this method is the situation where more than one arc goes from one state to another. If the first method is used, an incorrect assignment may occur very easily due to the small difference in distances between various arcs. In this editor, the arcs are highlighted in turn and the user is asked to choose the desired one.

For the arc parameters, both input tokens and diagram names can be used in the input token field. In order to differentiate between them, the diagram names are quoted with brackets. For example, a call to subdiagram "diagram1" is represented as "(diagram1)" in the arc parameter.

4.6. Data Structure

The data structures used in the editor are shown in Figures 4.4 and 4.5. In the group data structure (Figure 4.4), each primitive in a group is represented by a frame number pointing to the RTN database. A pointer instead of duplicate copy of the primitive record is used to reduce memory requirement. The basic structure of each element consists of two fields, one is a pointer to the RTN database and the other is a

pointer to the next element. All primitives of the same type are grouped together and a header record is used to represent this group of primitives. Four types of primitives (state, state name, arc and arc parameter) are used in "grouping", so at most four linked lists of elements are formed and their header records are linked together. These header records have three fields, element type (Frame type)[16], pointers to next header record (Next) and the elements (Element). A group record is used to represent each group. Each record holds the name of the group (Group Name), a pointer to another group (Next) and a pointer to the first head record (Element). This data structure has the advantage of decreasing execution time. Whenever a group is inserted into a diagram, the "state" type is searched first and displayed followed by "arc" and "arc parameter" types. If all the records are simply linked together, it is necessary to search through the whole database to sort out all the elements of a particular type.

For the diagram data structure shown in Figure 4.5, a linked list is used in a similar way to join all the diagrams. Within each diagram header record, a name field (Network Name) is used to hold the diagram name and pointers are used to link state list (State), arc list (Arc), arc parameter list (Para) and point to the next diagram (Next). The state pointer points to a list of state records. Each record contains the name of the state (State Name), the x, y coordinates of the state (Centerx, Centery) and a pointer to the next state (Next). For the arc record, the head state (From State), the tail state (To State), the four coordinates of the Bezier curve (Xpos, Ypos), a pointer to the associated arc parameter (Para) and a pointer to the next record (Next) are used. The arc parameter records hold the starting x, y coordinates and the names of the input token (Input Name, Inxpos, Inypos), the output token (Output Name, Outxpos, Outypos) and the application procedure token (Appl Name, Appxpos, Appypos). A pointer to the next record (Next) is included as well.

[16] For the discussion in this section, the names quoted inside the blankets "()" are names used in the data structure diagrams.

Pointer structures are used for all the data structures in the editor. This structure facilitates the insertion and deletion of records and is flexible in size.

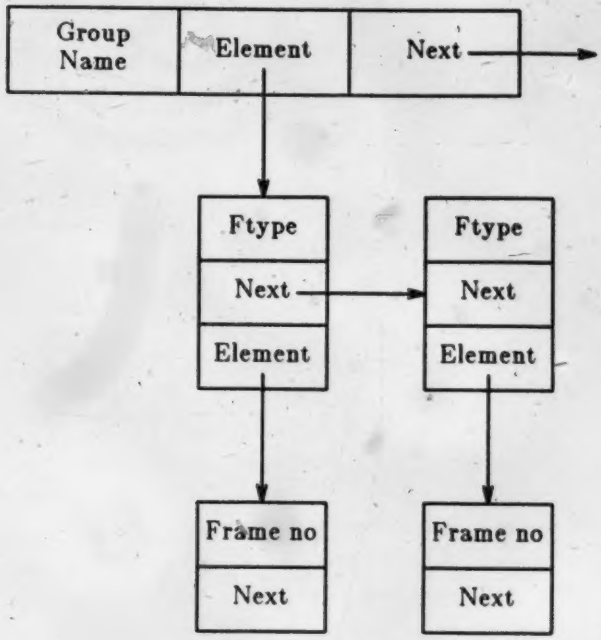

Figure 4.4 The Data Structure of Group used in the Editor

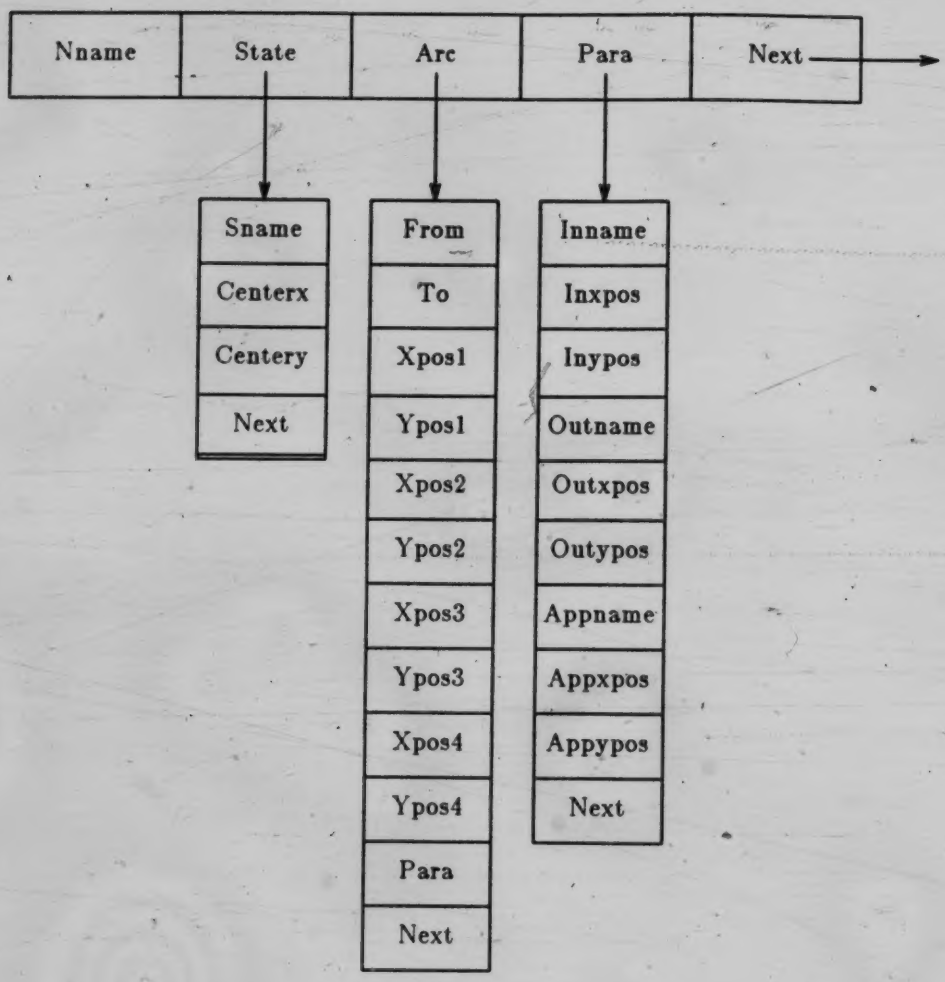

Figure 4.5 The Data Structure to Store the RTN in the Database

4.7. Program Flow

The organization of the program uses the basic structure of Windlib and is divided into several procedures which act as event handlers for the windows. The editor is divided into 4 windows, so four event handlers are used. The use of the handlers for "message" and "current selection" windows is very limited. They send an event to the work window when the user enters data into these windows[17]. The handler for the

[17] The only situation in which input is send to the "message" and "current selection" window is

"menu" window is used to select commands and primitives and highlight the options that have been chosen. The actual execution of the editor is from the work area event handler and it is the main control of the editor. The logic flow of the program is as shown below.

(1) The work area event handler receives an event from the input source or from another handler.

(2) Depending on the current command the editor is using, the work area passes the event to the appropriate procedures, such as

   (a) Add command procedures.

   (b) Move command procedures.

   (c) Delete command procedures.

   (d) Rename command procedures.

   (e) Next network procedures.

   (f) Exit command procedures.

   (g) Group command procedures.

(3) Each of these procedures gets the event and processes it according to the current object selected and updates the screen and database.

(4) Control is then passed back to the work area event handler to wait for next event.

(5) If during the processing of an event, an error is detected, the error handler is invoked to print an error message, the editor then waits for the user to reenter the correct input or enter a new command.

---

when the user enters the state name or arc parameter while the tracking cross is in these windows. This explains why the input is sent to this window.

4.8. Example

Figure 4.6 Initial Layout of the Screen

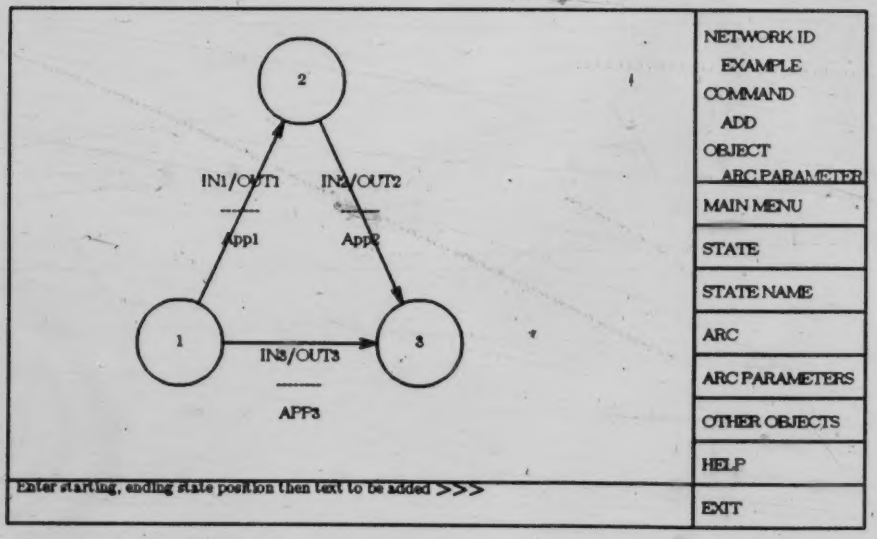

Figure 4.7 The Screen Layout After a Diagram is Created

In this section, an example of how to create an RTN using the graphical editor is shown. Although the diagram is very simple, it shows the usage of various commands

in the editor. The following is a step by step illustration of how the diagram is created. Figure 4.6 shows the initial layout of the editor and Figure 4.7 shows the final appearance of the screen after the diagram is created. The sequence is:

[1] The user puts the tracking cross on the "OBJECT MENU" menu item and presses the left button on the tablet[18] to select the object menu as shown in figure 4.7.

[2] The tracking cross is then placed in the work area and the user presses the same button again. A circle is drawn on the screen with the tracking cross positioned at the center[19].

[3] Three circles are drawn in this way.

[4] The user then selects the "STATE NAME" menu option to enter the state names and another prompt message "Place the cross inside the state then enter the name >>>" is displayed.

[5] The state name can then be entered by putting the tracking cross inside the corresponding circles (without pressing any button) and entering "1", "2" or "3" with a carriage return. The names will then be placed at the centers of the circles (the states).

[6] The "ARC" option is selected to enter the arcs and the editor requires the user to first specify the tail and the head states and then two points on the path by pressing the button when the cross is in the appropriate places.

[7] Three directed curves are created in this way by specifying state 1, 2 and 1 as the tail states and state 2, 3, and 3 as the corresponding head states.

[8] The "ARC PARAMETERS" option is then chosen and the editor responds with the message "Enter tail, head state, starting position then text to be added

---

[18] The left button is used to generate an event of type 1. Any tablet or mouse can be used provided that they are capable of generating events of type 1 and 3.

[19] The default command of the editor is "ADD" and the default primitive is "STATE"

>>>". The user is required first to specify the tail state, the head state of the corresponding arc and then put the tracking cross on the position for the arc parameter.

[9] To enter the arc parameter $\frac{In1/Out1}{App1}$, the user places the tracking cross inside state 1 and presses the left button on the tablet (to specify the tail state) then repeats the same operation for state 2 (to specify the head state). The tracking cross is then placed in the position which will become the lower left corner of the first character for the input token field (no pressing of button is required). The arc parameter is then entered from the keyboard in the format "In1/Out1/App1" followed by a carriage return.

[10] The same procedure is repeated for the other two arc parameters.

[11] The user can then choose the option to exit the editor or create/edit another diagram.

Chapter 5

The Event Handler Generation Algorithms

5.1. Introduction

When recursive transition networks are entered with the graphical editor, all the data is stored in a database. A code generator extracts the necessary information from the database for assembling the event handlers. The generated code then goes through an assembler and a "C" language compiler resulting in object modules that are linked with other modules (i.e. the presentation component and the application interface model) to form a complete UIMS. The whole process is shown in Figure 5.1.

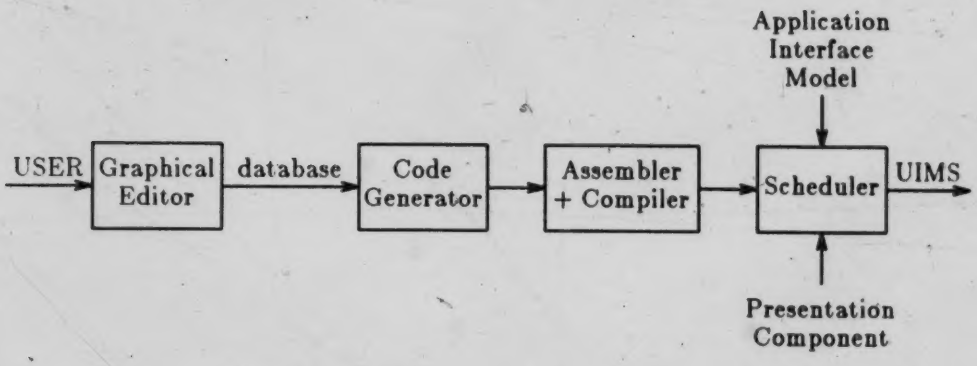

Figure 5.1 Code Generation Process

The run time execution of the UIMS is controlled by the scheduler. Its main functions are processing the events and organizing the event handlers' activities. This chapter describes the design of the code generator, its requirements and the algorithms used. The requirements of the generator, besides generating code, are detecting and reporting both logical and syntactical errors. The code to be generated is in an Event Based Internal Form (EBIF). Before any code is produced, the Leading Relations for all the recursive transition networks must be calculated. The format and meaning of the EBIF as well as the definition and the calculation of the Leading Relation are

described in this chapter.

5.2. Description of Event Based Internal Form

The EBIF, designed by Green [Green84e], consists of a "C" programming language procedure representing the event handler and various tables containing information used by the UIMS scheduler at run time. This data is extracted by the assembler to build the scheduler's tables. Figure 5.2 outlines the structure of an event handler. Basically it is separated into two components, the first part contains the tables and the second part, quoted between "%", is the "C" programming language representation of the event handler.

The EBIF language is an event driven language[20] and every operation is based on events. The event handlers communicate with each other by events. Tokens, on the other hand, are used to communicate with the application interface model and the presentation component.

```
event_handler
a, b, c
event1, event2, event3, event4
token1 event1
token2 event2
token3 event4
%
event_handler( d, e, f, g )
int d, e, f, g
{
  .
  .
}
%
```

Figure 5.2 Example of EBIF Event Handler

The EBIF event handler looks like an ordinary procedure with the first line having its name. The second line contains three numbers corresponding to the number of

[20] An Event driven language is a language which describes its operations by means of events and event handlers. When an event is generated, it is sent to an event handler, where the statements that process the event are executed.

variables, tokens and events used by each instance[21] of the event handler. The third line is a list of events declared in the event handler. Following it is a token table used by the scheduler, each line holds a token name and its equivalent event name. These few lines form the first part of the event handler description and provide sufficient information for the assembler to build the scheduler tables.

The second part of the event handler is the "C" programming language routine which is extracted by the assembler and compiled to form an object module. The name of this procedure is the same as the event handler name. It has four parameters representing the instance number of the handler, the event (which is an integer), the event value, and a pointer to the local variables of the event handler. The main body of the procedure consists of a case statement with each event represented by a separate case. If the event handler is active and an event which matches one of its cases arrives, the UIMS scheduler will execute the statements indicated by that event.

5.3. The Leading Relation

The meanings of the Leading Relation for RTNs and context free grammars are similar and the procedures used to calculate them are basically the same. Below is a general description of its meaning and how it is calculated.

5.3.1. Definition of the Leading Relation

The formal definition of the Leading Relation for the subdiagrams can be found in [Green85b] which states:

> Let Σ be the set of all the possible input symbols for the subdiagram and S be a string of symbols from Σ.
> For a subdiagram, d, let L(d) stand for the set of strings in Σ that are recognized by d. That is, every string in L(d) labels a path from the initial node to the final node of d. The relation Leading is defined in the following way
> $$\text{LEADING}(d) = \{ a \mid a \in \Sigma \text{ and } aS \in L(d) \}$$
> The relation LEADING for a given subdiagram is the set of input symbols

[21] When an event handler is created during execution, a new instance of the event handler is created.

that subdiagram is expecting when it is invoked.

This relation ensures that a subdiagram is called only when a proper set of input tokens is entered. Figure 5.3 shows a situation in which "token1" is entered into the system. If the arc labeled by "subdiagram" (not having token1 in its Leading Relation) is invoked first, no match will be found in its token list and the error handling procedure will be called to handle this unmatched token. This behavior is undesirable.

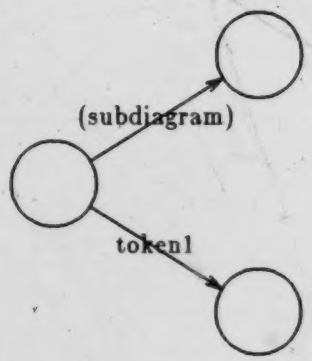

Figure 5.3 Example Showing the Importance of the Leading Function

5.3.2. Calculating and Checking of the Leading Relation

The method used to calculate the Leading Relation is relatively straightforward. Every arc in the subdiagram is scanned in turn and all the arcs having the initial state at their tails are marked. The Leading Relation is the set of all the input tokens on these marked arcs. This result is stored in a separate file for each subdiagram. The only pre-requisite in this calculation is the existence of a unique initial state for each subdiagram. Two types of parameters can be found in this relation file, input tokens and subdiagrams names. They are stored in the same Leading Relation file but with different formats to differentiate between them.

The checking of the Leading Relation can be done either at execution or code generation time. It is found that doing the job at run time will decrease the execution

speed and is less efficient. On the other hand, more time is spent on code generation as one separate pass is devoted to scan through all the subdiagrams in order to extract information for building the Leading Relation file. Since run time execution is a real time process, the first approach was selected. Hence the Leading Relation needs to be calculated before any code is generated.

If an entry in the Leading Relation of a subdiagram (e.g. subdiagram1) is a token name, this is a token that can call this subdiagram. On the other hand, if the entry is a diagram name (e.g. subdiagram2), the valid token names to call the first subdiagram (i.e. subdiagram1) are the tokens found in this called subdiagram's (i.e. subdiagram2) Leading Relation file. A recursive call may be required to search for these token names. In this case, the input token searching expands to a tree searching problem. A problem occurs when a subdiagram on an arc is a parent of the current diagram, this is shown in Figure 5.4. In this figure, subdiagram "D1" can jump to "D2" which can in turn jump to "D4". If the Leading Relation set of "D2" contains "D4", and the Leading Relation set of "D4" contains "D1", then when the Leading Relation set of "D1" is calculated, it will check for the Leading Relation set of "D2". Since "D4" is an element in this file, the Leading Relation set of "D4" is searched. An infinite loop will develop in this situation because the set for "D4" contains a call to "D1".

A way to avoid this is to construct a table containing the names of all the diagrams that have already been searched and every time a new diagram's Leading Relation is required, this table is checked first. If the name is in the table already, then the particular subdiagram does not need to be searched.

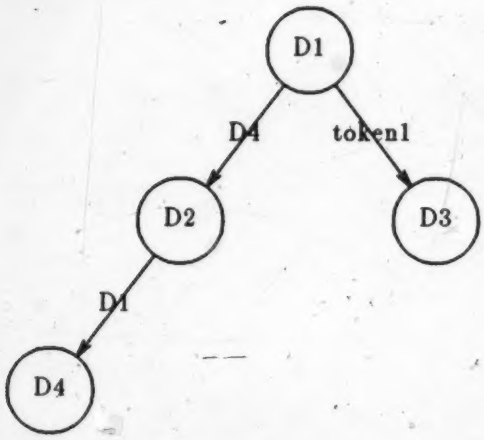

Figure 5.4 Example of Infinite Searching in the Leading Relation

5.3.3. Examples

Two examples are shown in this section to illustrate how the Leading Relation files are formed and how to calculate the valid tokens that can invoke particular diagrams.

5.3.3.1. Example 1

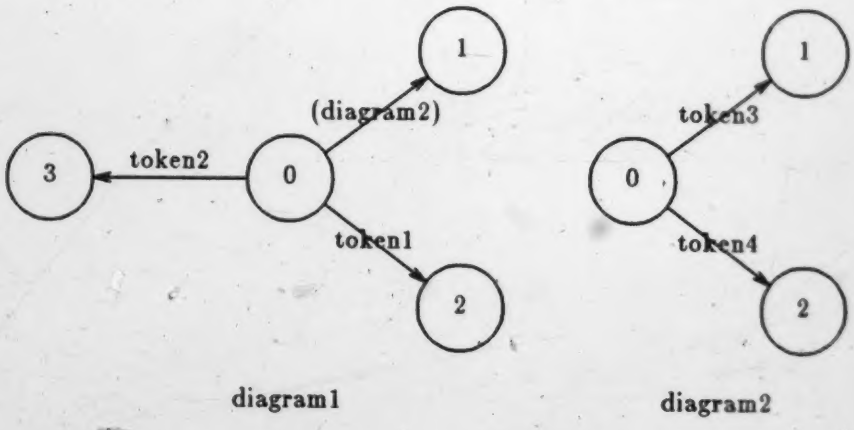

Figure 5.5 Example Showing How The Leading Relation Files are Formed

In this example, the Leading Relation set for diagram1 is token1, token2 and diagram2; while the Leading Relation set for diagram2 is token3 and token4. In order to differentiate between tokens and diagrams in the Leading Relation files, diagram names are all quoted with brackets. So the Leading Relation files for diagrams 1 and 2 are:

diagram1	diagram2
token1	token3
token2	token4
(diagram2)	

The tokens that can call diagram2 are token3 and token4; on the other hand, token1, token2 and the tokens in Leading Relation set of diagram2 are the tokens that can invoke diagram1. Hence, its tokens are token1, token2, token3 and token4.

5.3.3.2. Example 2

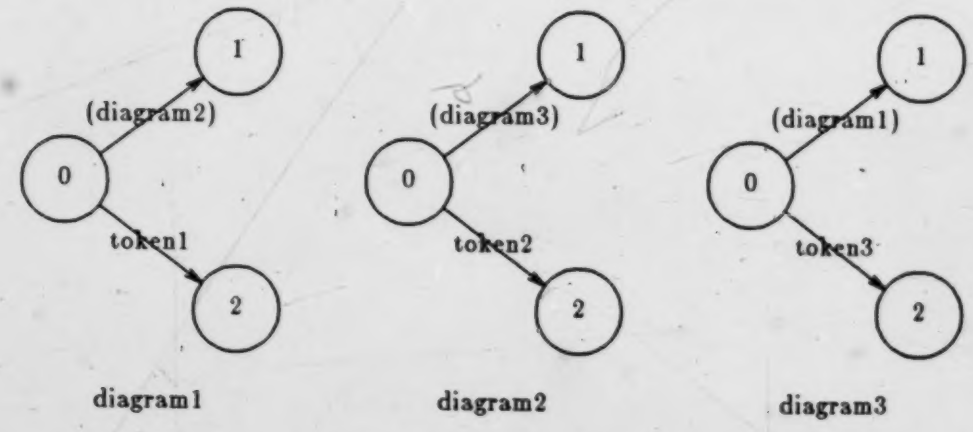

Figure 5.6 Example showing Recursive Calls of the Leading Relations

Similar to example 1, the Leading Relation files of the three diagrams are:

diagram1	diagram2	diagram3
token1 (diagram2)	token2 (diagram3)	token3 (diagram1)

The legal set of tokens for diagram1 are token1 and the legal set of tokens in diagram2. Similarly, the set of tokens for diagram2 and diagram3 are token2 and token3 with the tokens sets in diagram3 and diagram1 respectively. As a result, a recursive dependency occurs. The table holding all the diagram names that have been searched already (as described in section 5.3.2) is a great help in solving this problem. The steps used for calculating the legal set of tokens for diagram1 are shown below. Basically a table is used which holds the set of all the tokens calculated and all the diagrams that have been searched already.

Step 1:

When the code generator starts to produce code for diagram1, the only token is token1 and it needs to search the Leading Relation file of diagram2. So a table, as shown below, is formed in which "token1" is the token to invoke diagram1 directly, and the diagram that has already been searched is "diagram1".

token	diagram searched
token1	diagram1

Step 2:

After searching the Leading Relation file of diagram2, token2 becomes a legal token as well. So the table has two entries for tokens. Since the Leading Relation file of diagram2 contains diagram3, the generator starts searching the Leading Relation set of diagram3.

token	diagram searched
token1	diagram1
token2	diagram2

Step 3:

token	diagram searched
token1	diagram1
token2	diagram2
token3	diagram3

After diagram 3 is searched, the legal tokens to call diagram1 are token1, token2 and token3. At this point, the system should go and search for tokens in the Leading Relation file of diagram1; however, it has searched this table already so the process stops here. Hence, the legal tokens for calling diagram1 are token1, token2 and token3. The legal sets of tokens for diagram2 and diagram3 are calculated in the same way.

5.4. Error Handling

Error handling is the first task which must be considered when code is generated. The fundamental requirement for the error handler is to detect as many errors as possible before they are found by the assembler, the compiler or the scheduler. A serious problem may occur if the errors are undetected at creation time. They may cause the program to run into an infinite loop or an execution error. The following is a list of all the common errors, both at creation and run time, and how they can be detected.

5.4.1. Types of Errors

The common types of errors that can be found while generating code are:

(1) No unique initial state for a subdiagram. If there is no initial state in a subdiagram, the scheduler cannot start the execution of the subdiagram when it is invoked. If more than one initial state exists, the scheduler does not know which state to start execution in.

(2) No terminal state exists in a subdiagram, which implies the program cannot finish executing the subdiagram and ends up developing an infinite loop inside that

diagram.

(3) The input parameter field of an arc connecting two states is empty. The event-handler generator cannot generate code for this arc since code production is based on this token name and type.

(4) The state names are not unique within each subdiagram. This may result in the wrong interpretation of the data by the code generation program. The generator treats the two distinct states with the same name as one state.

(5) Two or more subdiagrams have the same name within a user interface. This gives rise to an ambiguity when this name is called by another subdiagram. The scheduler cannot determine which subdiagram is the correct one. This error can be detected by the "C" compiler later in the process because two procedures are using the same name within a program.

(6) A subdiagram called by an other subdiagram does not exist in the system. Same as (5), the loader will report the error of non-existing procedure being found when all the subdiagrams' object modules are linked together.

One of the principles for error handling used in the generation program is to find the errors as early as possible. The generator will then stop generating code. However, some errors, such as (5) or (6) above, are impossible to detect until the whole user interface is assembled unless some dedicated and time consuming procedures are used. For example, the error of non-existing diagram name (error #5 above) can be detected if a separate procedure is used to scan through and record all the diagram names. However, this is a time consuming process because the whole database needs to be scanned through once in order to get all the diagrams names and build a table to hold all the names.

5.4.2. Algorithms

The algorithms discussed below are aimed at solving problems (1) to (4) stated above. In order to facilitate the detection of errors and the generation of code, two main tables are built. The first table, called the state table is formed by scanning through individual subdiagrams in the database. It contains the names of all the states in each subdiagram. The arc table is another table built by scanning through all the arcs in the subdiagram and contains the input token name, the output token name, the procedure name, and the head and the tail state names of the arc. It also holds a unique event number for each token. This number is produced by the code generator and is used in code generation.

The checking of unique initial state is done by going through the state table. If the state name called "0", which is assumed to be the starting state name, is not found the diagram has a syntax error. Duplicate state name checking is done when the state table is built. Whenever a new state is added to the table, the whole table is scanned through once. If the name already exists, a duplicate name is being used within a subdiagram. When the arc table is built, the third error can be detected. Whenever the code generator detects a missing input token in the arc parameter, the error is immediate reported. To detect the error of non-existing ending state, more work needs to be done. Another table, the ending state table, containing all the ending state names is built. Every state in the state table is compared with the arc table, if the state never appears at the head of an arc, it is an ending state and should enter the ending state table. After processing all the states in the state table, if the ending state table is still empty, then an error of non-existing ending state should be reported. Figure 5.7 summarizes the contents of all these tables.

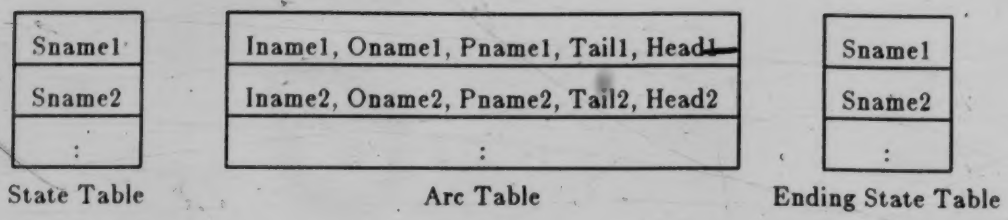

Figure 5.7 Tables of Contents of Conversion Program

5.4.3. Example

Using the example1 in section 5.3.3, the following tables are created for diagrams 1 and 2 respectively. Since the arc table needs further modification as described below only the state tables and the ending state tables are shown.

State Table	Ending State Table
0	1
1	2
2	3
3	

Figure 5.8 The State and Ending State Table of Diagram 1

State Table	Ending State Table
0	1
1	2
2	

Figure 5.9 The State and Ending State Table of Diagram 2

5.5. Principles of code Generation

The format of the EBIF event handler has already been described in section 5.2. The algorithms for code generation are divided into two parts. The second part of the event handler is generated first because it produces results which are used in the construction of the first part of the event handler.

Since the EBIF event handler is an event driven system, it will execute different sets of commands depending on the input event. Also the set of arcs that can be traversed at any time depends on the current state of the RTN. When the event handler is produced these two control variables (i.e. the current state and the generated event) are used to select the statements for execution. Two switch statements are used to control the execution of the event handler. The control variable "event" selects the commands that are executed when an event arrives. An event that resides on several arcs can call different sets of commands depending on the current state of the RTN (and hence the event handler) so a switch statement is constructed within each "event" case. The following three sub-sections give the general design principles for code generation and how the second and first parts of the EBIF event handler code conversion program are designed.

5.5.1. General Design Principles

Although a complete user interface consists of a number of event handlers, they can be entered into the system with the graphical editor at the same or at different times. The code generator is designed to create separate files for individual event handlers. This gives the user interface designer the capability to build and use libraries of event handlers. Another advantage of considering each event handler as a separate unit is to ease the modification and debugging of the event handlers. The scheduler needs to link all the event handlers together to form a complete unit. These handlers can be in two formats, either in the RTN format that must be processed by

the code generator, or in the EBIF. This makes it possible to use a library of the commonly used event handlers in both RTN and EBIF formats.

Whenever an event arrives, the event handler compares it with all the events that it can process. If a match is found, the corresponding set of statements is executed. In the case of no match, the system prints an error message and asks the user to reenter the token until a match is found. This error handling method is the simpliest and the easiest approach. Further extensions of the system may lead to more sophisticated error handling procedures.

The sequence of execution for the parameters on an arc is first the application procedure then the output token. By fixing the sequence of execution for the parameters on the arcs, the conversion step is easier. Different results may be obtained if the two operations are processed in different orders. The user interface designer can now be sure that the code generator interprets the RTNs in the same way as they are designed by the user interface designer. The application procedure or the output token field or both can also be left empty, in which case the event handler assumes they are not used.

The input token field of any arc can be the name of an input token or a subdiagram. In the case of a subdiagram name on the arc, control will pass to the subdiagram. The program will continue execution there until the end of the diagram. The name of the caller and the head of the current arc must be passed to the calling subdiagram in order to restore the return address.

5.5.2. Generating the C Procedure

The basic approach to generating the "C" procedure is to generate the two switch statements described above. The name of this procedure is very simple, it uses the same name as the event handler, but in order to differentiate it from other system procedures, "eh" is added at the front.

5.5.2.1. Generating the Tables

Three tables, the state table, the ending state table, and the arc table, as described in section 5.4.2, are required for code generation. In fact, the arc table is divided into two parts, a token table and a diagram table. If the input token field of the arc parameter is a token name, it is put into the token table. On the other hand, if a diagram name is found there, the arc parameter is put in the diagram table. The respective fields of the tables are shown in Figure 5.10. Two tables instead of one are used in generating code because all the entries in the diagram table will be preprocessed to form entries for the token table before the code is generated. A detailed description of this process is presented in the following sections.

token table	diagram table
input token	
diagram name	diagram name
output token	output token
procedure name	procedure name
tail	tail
head	head
event#	event#

Figure 5.10 Token and Diagram Table

For the diagrams shows in example 1 above, the token and diagram tables are as shown below:

Token Table						
Input Token	Diagram Name	Output Token	Procedure Name	Tail	Head	Event#
token1				0	1	event0
token2				0	2	event1

Diagram Table					
Diagram Name	Output Token	Procedure Name	Tail	Head	Event#
diagram2			0	3	event2

Figure 5.11 Token and Diagram Table of Diagram1

Token Table						
Input Token	Diagram Name	Output Token	Procedure Name	Tail	Head	Event#
token3				0	1	event0
token4				0	2	event1

Figure 5.12 Token and Diagram Table of Diagram2

In generating the "C" code for the event handler, all four tables are used and the diagram table is the first to be considered. The diagram table contains the names of all the diagrams found on the arcs. The recursive search algorithm of the Leading Relation as described before is used to find all the valid tokens on which calls of subdiagrams are possible. Each token that is found is put into the token table along with the output token, the procedure name, the tail and the head states copied from the diagram table. So all the tokens found from the same Leading Relation file will be put into the token table with the same procedure name, output token name, tail and head states as in the diagram table. In addition, they will include the diagram names in their records. Figure 5.13 shows the resulting arc table after the preprocessing has finished[22].

[22] Since diagram2 does not call other subdiagrams the diagram table is not formed and no conversion is required.

token table						
input token	diagram name	output token	procedure name	tail	head	event#
token1				0	1	event0
token2				0	2	event1
token3	diagram2			0	3	event2
token4	diagram2			0	3	event3

Figure 5.13 The Result of Combining Token and Diagram Tables for Diagram1

The code is actually generated from the input token table. Each entry is considered in turn and each input token is considered to be separate case. A token can invoke different sets of statements depending on the current state of the event handler, so another set of case statements is used. The format of the generated code is:

```
case (event) : switch (state)
        {
          case :
                .
                .
                .
          case :
        }
```

Two special conditions need particular attention in generating code. They are:

(1) the head of an arc is the ending state of the subdiagram.

(2) the input token field of an arc is a subdiagram name.

Three different situations arise from the above combinations and they need to be considered individually. These cases are:

(1) the arc is not a subdiagram call and the head state is not an ending state.

(2) the arc is not a subdiagram call but its head state is an ending state.

(3) the arc is a subdiagram call.

The conversion algorithm in the first case is quite simple. The statements include:

(1) If the application procedure field is not empty, a "send-token" command is created to send the procedure name as a token to the application interface model.

(2) If the output token field is not empty, the output token is sent to the presentation component.

(3) Command to change the event handler state to the head of the arc.

In the second case, in addition to all the statements discussed above, some methods must be used to return control to the calling subdiagram. The technique used is to send a special event called "continue" back to the calling program to indicate the termination of the subdiagram's execution. Besides the diagram name, the tail state of the arc which called the subdiagram is sent back as well to restore its state. The last operation is to signal the scheduler to kill the event handler instance that has just exited from the terminal state.

For the last case, the problem is divided into two parts as shown in Figure 5.14.

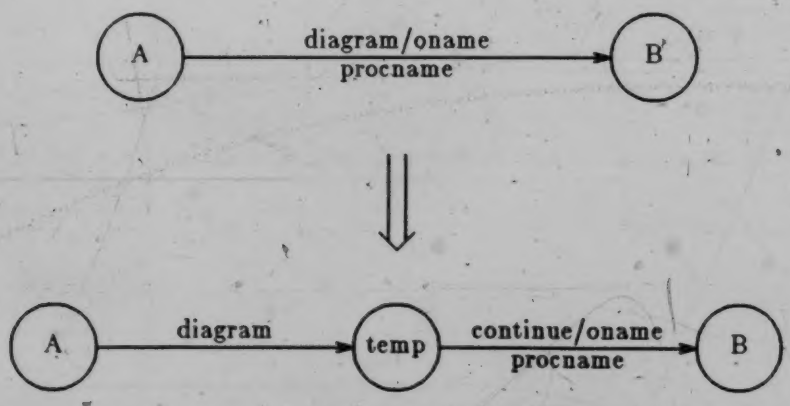

Figure 5.14 Generating Code Algorithm With Arc For Diagram Call

The main reason for doing this splitting is that the tokens (application procedure and output token), are not sent to the application interface module and the presentation component until after the subdiagram is processed. If the returned state is the head of the arc already, these two tokens will not be processed because the arc holding

them has already been traversed. A temporary state[23] is created for this purpose which serves as the return state of the subdiagram. The application procedure and the output token are assumed to be on the arc from the temporary state to the head state. This handles the calling of the subdiagram and processing the procedure and output token. When the called subdiagram exits, an event called "continue" is generated. This will activate the second arc, and the output token and the application procedure name are sent to their destinations.

The algorithm for doing the conversion is to first consider the calling of the subdiagram by creating an instance of the subdiagram and send an event to that instance with the temporary state and the caller instance name as two variables. The second step is to add the head state (state "B"), the tail state (state "temp") and the arc's parameters to the token table with the event name "continue".

5.5.2.2. Examples

By using the algorithms as described above and the two arc tables in the previous section, the following two "C" procedures are generated.

```
ehdiag1 ( iname, ename, value, variables )
  int iname;
  int ename;
  int value;
  int variables[ ];
{
  switch ( ename )
  {
    case ( event0 ) :
      switch ( state )
      {
        case ( 0 ):
          state = 2;
          send_event( variable[ 0 ], continue, variable[ 1 ] );
          send_event( variable[ 0 ], ename, 0 );
          destroy( iname );
          return( );
      }
    case ( event1 ) :
      switch ( state )
      {
        case ( 0 ):
```

[23] The temporary states have the names "temp" + 0,1,2... to handle more than one subdiagram call within a subdiagram.

```
        state = 3;
        send_event( variable[ 0 ], continue, variable[ 1 ] );
        send_event( variable[ 0 ], ename, 0 );
        destroy( iname );
        return( );
      }
     case ( event2 ) :
      switch ( state )
      {
       case ( 0 ):
        create_instance( diag2, 2, this_instance, temp0 );
        send_event( new_instance, ename, value );
        state = temp0;
        return( );
      }
     case ( continue ) :
      switch ( state )
      {
       case ( temp0 ):
        state = 1;
        send_event( variable[ 0 ], continue, variable[ 1 ] );
        send_event( variable[ 0 ], ename, 0 );
        destroy( iname );
        return( );
      }
     case ( event3 ) :
      switch ( state )
      {
       case ( 0 ):
        create_instance( diag2, 2, this_instance, temp0 );
        send_event( new_instance, ename, value );
        state = temp0;
        return( );
      }
   }
}
```

Figure 5.15 The 'C' Language procedure of the EBIF for Diagram1

```
ehdiag2 ( iname, ename, value, variables )
 int iname;
 int ename;
 int value;
 int variables[ ];
{
 switch ( ename )
 {
  case ( event0 ) :
   switch ( state )
   {
    case ( 0 ):
     state = 1;
     send_event( variable[ 0 ], continue, variable[ 1 ] );
     send_event( variable[ 0 ], ename, 0 );
     destroy( iname );
     return( );
   }
  case ( event1 ) :
   switch ( state )
   {
    case ( 0 ):
     state = 2;
```

```
        send_event( variable[ 0 ], continue, variable[ 1 ] );
        send_event( variable[ 0 ], ename, 0 );
        destroy( iname );
        return( );
    }
  }
}
```

Figure 5.16 The 'C' Language procedure of the EBIF for Diagram2

5.5.3. Generating the Event Handler Tables

The token and event tables in the EBIF event handler are generated by going through the input token table once. When this table is created, every time a token is added, a corresponding event is created and inserted into the table, so the token and event tables can be extracted from the input token table. The event list is created in a similar way with the exception that another event called "continue" is included in the list as well. This event is used to pass system control to and from subdiagrams.

The three numbers for the event handler which represent the number of local variables, events and tokens are relatively straightforward to generate. The number of local variables should be three corresponding to the instance and the state number of the calling event handler and also a variable called "state" which represents the current state of the RTN. The total number of tokens equals the number of input tokens.

5.5.4. Examples

The event handler tables for the example 1 are shown in Figure 5.17 and 5.18 respectively.

```
ehdiag1
2 5 5
event0 event1 event2 continue event3
token1 event0
token2 event1
token3 event2
continue continue
token4 event3
```

Figure 5.17 The Event Handler Table for Diagram 1

```
ehdiag2
2 2 2
event0 event1
token3 event0
token4 event1
```

Figure 5.18 The Event Handler Table for Diagram2

Chapter 6

Implementation of the Code Generator

6.1. The Environment

The generator is programmed in the "C" language under the UNIX[24] operating system and runs on a VAX[25] 11/780. A large portion of the code is machine independent due to the portability of the "C" language and the use of commonly available UNIX system commands only. In order to make the generated code easier to read, it is formatted. The system command "indent" which automatically formats a "C" program is not used in the generator since it is not commonly available on all UNIX-like operating systems which support "C". The generated code is written onto a standard ASCII file, and a database called "FDB", as described in Chapter 4, is used as input.

6.2. Strengths and Weaknesses

The generator is made up of two different modules which are:

(1) A procedure to generate tables by extracting data from the database. Leading Relation files are built in this stage and all possible errors are checked.

(2) A procedure, which can assume perfect input data, to generate the EBIF event handler code.

There are two main advantages of splitting the program into two separate modules. First, when the EBIF format is modified or extended in the future, only the procedure used to generate code needs to be modified. On the other hand, when a new or modified version of the database is used, only the first part needs to be rewritten. On the whole, the program is easy to modify when any part of the early design is changed.

---

[24] UNIX is a trademark of AT & T Bell Laboratories, Inc.
[25] VAX is a trademark of Digital Equipment Corporation.

There are some weak points in this generator, solutions are available to solve these problems but they were not used in order to simplify coding or decrease the computation time required. The event handler uses nested switch statements to select the statements to execute when an event arrives. Each entry in the input token table will generate a case statement based on "event" and each record associated with the input token will generate a "state" case. The first switch statement is unavoidable but the second one is not required and an "if" "then" statement may do a better job if the token appears only on one arc. The other major weakness is the redundancy of code within each event handler. The same output token and application procedure name may be used on many arcs within a transition diagram, i.e. different combinations of events and states may include the same procedures to handle the output token and application procedure. The program should be able to detect this redundancy and combine them together to decrease the size of the object module.

6.3. Data Structure

The data structure used by the generator consists of a number of tables. They are implemented by linked lists in order to provide variable table sizes and facilitate deletions and additions. Two main linked lists are used for the input token and the diagram tables. The structures and arrangements of the linked lists are shown in Figure 6.1-6.4. Two other tables are used to hold the state names and the names of the end states in each diagram.

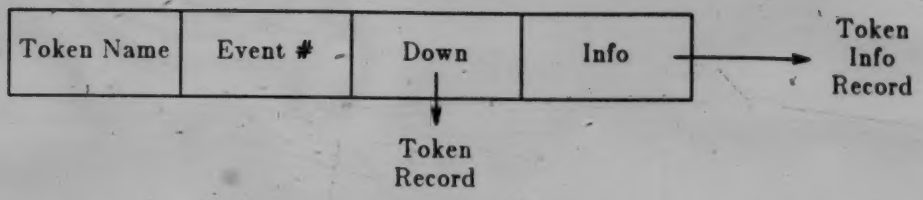

Figure 6.1 Token Record Structure

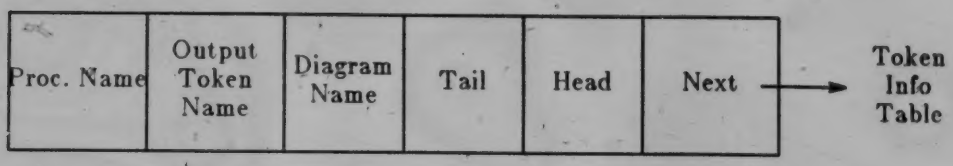

Figure 6.2 Token Information Record Structure

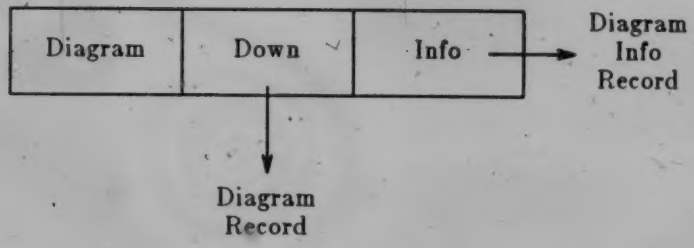

Figure 6.3 Diagram Record Structure

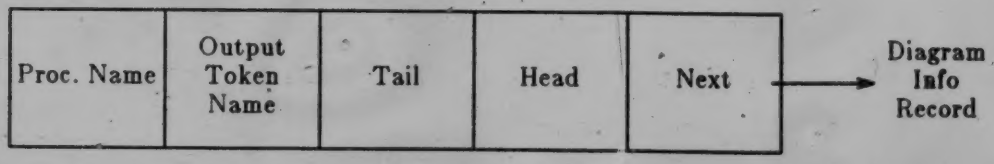

Figure 6.4 Diagram Information Record Structure

Each input token record holds a token name and an event number given by the generator to represent the token. The naming of these events is very simple. All of them

have the format "event" followed by a number which is incremented by one every time a new event is generated (The name of the first event is "event0" followed by "event1", "event2" ...). The diagram table is created in a similar way but without the event name. Each diagram record is divided into two input token records, as discussed in Chapter 5, when code is generated. The names of these tokens are the token name in the diagram record and a dummy name. This dummy name is not used in generating code except to signal the generator to produce an event called "continue" instead of an ordinary event number. All information records hold the application procedure and the output token names as well as the tail and the head states of the arc. A pointer is also available to point to the next information record if required, i.e. pointing to the next arc which has the same input token.

6.4. Program Flow

Two separate files are created for generating the code. They are used to store the code for the first and the second parts of the event handler. As described before, the second part is created first and some of the generated information is used to produce the first part. The sequence for generating code is:

(1) Open the database which contains the data.

(2) Use the data from the database to form the tables shown in Figure 6.1-6.4 and also the two state tables.

(3) Check for possible syntax errors. If an error is found, report to the user interface designer and exit from the generator.

(4) Procedure to generate the procedure name which is "eh" + name of the diagram and also its parameter declarations. This generated procedure is used by the assembler later.

(5) Process the diagram table by generating two tokens for each diagram name.

Insert these two token names and their records into the token table.

(6) Assemble the code from the token table by generating a case statement for every event used.

(7) Within each event case statement, another switch statement is generated for states. Every information record will be treated as a separate case.

The above 7 steps generate the code for the "C" procedure of the event handler. The following two steps will produce the EBIF event handler's event list and the token-event table. A complete event handler is formed by combining these two parts.

(1) The first statement generated is the event handler's name which is the same as the procedure's name generated in first stage.

(2) The three variables in the event handler are calculated during code generation in the previous stage. The number of variables used is always 3, corresponding to the caller's name and its head state when a subroutine call is initialized. The last variable is the internal state of the event handler. The second number, the number of events processed, is the number of elements in the event list. The total number of tokens send from the presentation component to the event handler is the number of tokens used.

Chapter 7

Conclusions

This study has shown that building an automatic generator for the dialogue control component is both feasible and desirable. The following is a description of the merits of the RTN graphical editor and the conversion program as well as some further extensions to the system.

7.1. Merits of the RTN Editor

The "software development cycle" for any piece of software consists of "requirements", "specification", "design", "implementation", "testing" and "maintenance" [Green81]. The implemented generator takes care of the last 4 phases. One approach to designing the dialogue sequence between the user and the computer, is to represent their interactions with state transition diagrams. With the help of the RTN editor, the state transition diagrams can be directly entered into the computer and the system automatically generates the required user interface code. The design and implementation phases are combined together and the user interface designer need not bother about implementation. Since he only needs to concentrate on design phases, the total time spent on these two phases will be reduced.

The system also makes the tasks of testing and maintenance easier because if there is a bug or the dialogue produced does not meet the requirements, it is easier to make changes on diagrams than on textual statements. If program maintenance is done by people other than the designer, then more time is needed to understand the program before modifications can be made. This can be reduced if the program is represented graphically instead of textually. As a conclusion, the implemented system simplifies the job of the user interface designer and lowers the cost and time spent on designing the user interface. Actually the system provides a software development environment for user interface design.

In representing the recursive transition networks, a hierarchical structure is used, this forces the user interface designer (either willingly or unwillingly) to split a complicated problem into levels. A direct consequence is the ease of understanding, designing, and maintaining the dialogue sequence program. A complicated design task can be decomposed into smaller modules. This allows different user interface designers to design separate parts, the only requirement is a tight interface between different layers and modules. This splitting has the same advantages as dividing a computer program into a hierarchical structure in design. As Smith stated [Smith82]:

> Creating something out of nothing is a difficult task. Everyone has observed that it is easier to modify an existing document or program than to write it originally.

Since many basic user interface modules are common in many applications, they can be reused and merged together (with or without modification) when a new user interface module is designed and hence the development cost is reduced.

The user interface is an important component whenever a piece of software is designed. As Simpson said [Simpson82]:

> If you fail to take people into account during design, then your machine (or system or program) may be difficult or impossible for people to operate.

When the RTN graphical editor was designed, human factors were the first consideration, i.e. the ease of usage by the user is the first main objective of the whole design. The users of the editor range from experts to novices, for example experts can disable some features of the editor that aim at teaching or prompting novice users how to interact with the system. The user interface in the editor helps both types of users to develop a good user model for themselves to work with. By having a self-developed model, the RTN editor's user can interact, work faster and less error prone in entering transition network diagrams into the database.

User tailorability is another main consideration in the design phase, although this

is not provided at the user level, all input errors made by the user are recorded in a logging file. It is hoped that by analyzing the feedback from the users and the logging files, the user interface designer can have a better understanding of the user's requirements and better user interfaces can be designed in the future. Another facility provided by the editor is allowing the user to define his/her own object types by making use of the "grouping" command. The main purpose of user tailorability is to avoid common errors made by users and to speed up the operations because no matter how general or powerful the editor is, it will never satisfy all its users.

As Smith said [Smith82], "Keep the program simple" is a main consideration in design, so when the editor's operators were designed, a few papers [Brown76, Pavlidis84, Singh83] were analysed and the editor commands are the necessary and useful ones without any obvious overlapping between them. By keeping the commands to a minimum, universal and non-overlapping as this editor does, it is easier and faster for the user to create his own model. The user can then use the editor in his/her own way. More important, it minimizes the human memory demands.

7.2. Special Features of the Code Generator

Nowadays, the user interface is an important component in application programs and some of its functions are common in many applications. Libraries can be built to provide some standard modules for the user interface designer. The code generator allows libraries in both RTNs format and EBIF to be linked with the system. This provides greater flexibility and freedom in using library routines. The user interface designer can easily modify a diagram in RTN to fit his own requirements and at the same time use the event handlers provided by other dialogue control component notations. For example, some interactions are easier to represent in the event language and conversion to the RTN is not required before it is linked with the other subroutines.

Since this is the first stage in the design of the EBIF event handler, the conversion program is designed to be flexible and modular to accommodate further extensions of the EBIF: When the EBIF format is modified, only small changes are required in the conversion program.

A lot of error checking routines have already been built. They cover most of the errors that we usually found. Moreover, more checking is possible and easy to fit into the system if required as described in Chapter 6.

7.3. Further Extensions

A few extensions are possible for making the implemented system a more sophisticated development tool. First, some mechanism allowing the users to define their own operations, e.g. to define an operation for searching through the whole database for a particular string. Also, although the "grouping" operation is a powerful extension of the editor, it only supports "addition" and "deletion", further development of the editor should support other commands such as "moving" as well. The menu is now on the right hand side of the screen, this will cause no trouble for right handed users, but for left handed users, it is more convenient to have it on the left hand side and actually, users should be able to define the locations of all windows/menus themselves.

Since the whole dialogue sequence is made up of a number of diagrams, it may be necessary to flip through the diagrams in order to examine them for debugging and changing. A further extension of the editor is to display several diagrams, with reduced size, on the screen at the same time.

A possible further extension for the code generator is to provide a better algorithm for handling unmatched inputs. The algorithm that is used now is a very simple method in which the system will sit there waiting for the user to enter one of the expected inputs[26]. This should be at least modified to the case where all legal inputs

[26] Although "wild_cards" is a possible solution, this needs to be specified explicitly every time it

are displayed for the user to select from in case an error occurs.

A logging file may also be used to record the time response and errors made by the user. This file is useful for two reasons. First, it indicates how well the recursive transition networks were designed by counting the number of errors the user has made. Second, this file can be used as a basis for comparing the descriptive power of all three dialogue control component notations.

Another extension to the generator is to provide some facilities similar to "makefile" of UNIX. Whenever some event handlers of the whole user interface are changed, the changes should be detected and the related files reassembled and recompiled. A difficulty occurs because the Leading Relation of a modified event handler will affect a number of other event handlers that depend on this relation. An extensive search of all the event handlers can be done to check if a file depends on this Leading Relation. If an event handler depends on a modified Leading Relation, recompilation is required. However, this is a time consuming process since all handlers need to be checked. Better algorithms need to be designed to do this job efficiently.

---

is used.

References

[Brown76]M. D. Brown and S. W. Smoliar, " A graphics Editor for Labanotation", *Computer Graphics*, Summer, 1976, pp.60-65.

[Buxton83]W. Buxton, M. R. Lamb, D. Sherman and K. C. Smith, " Towards a Comprehensive User Interface Management System", *Computer Graphcis*, July, 1983, pp.35-42.

[Edmonds81]

E. Edmonds, " Adaptive Man-Computer Interfaces", in *Coombs and Alty (ed.) Computer Skills and the User Interface, Academic Press*, 1981, pp.389-426.

[Green81] M. Green, " A Methodology for the Specification of Graphical User Interface", *Computer Graphics 15*, 3 (August, 1981), pp.99-108.

[Green84a]M. Green, *FDB: A Frame Based Database System*, Department of Computing Science, University of Alberta, Edmonton, Alberta, September, 1984.

[Green84b]M. Green, " Design Notations and User Interface Management Systems", *in [Seeheim84]*, 1984.

[Green84c]M. Green, " Report on Dialogue Specification Tools", *Computer Graphics Forum 3*, 4 (1984), pp.303.

[Green84d]M. Green and N. Brigdgman, *WINDLIB Programmer's Manual*, Department of Computing Science, University of Alberta, Edmonton, Alberta, September, 1984.

[Green84e]M. Green, The University of Alberta User Interface Management System Design Principles", Human-Computer Interaction Project Report #1, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, 1984.

[Green85a]M. Green, " The University of Alberta User Interface Management System", *Siggraph'85 Proceedings*, 1985.

[Green85b]M. Green, User Interface Models", Human Computer Interaction Project Report #2, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, 1985.

[Kaisk82] D. J. Kaisk, " A User Interface Management System", *Computer Graphics*, July, 1982, pp.99-106.

[Newman68]

W. M. Newman, " A System for Interactive Graphical Programming", *Proc. 1968 Spring Joing Computer Conference*, Washington D.C. , 1968, pp.47.

[Olsen Jr.83a]

D. R. Olsen Jr. and E. P. Dempsers, " SYNGRAPH: A Graphcial User Interface Generator", *Computer Graphics*, July, 1983, pp.43-50.

[Olsen Jr.83b]

D. R. Olsen Jr., " Automatic Generation of Interactive Systems", *Computer Graphics*, January, 1983, pp.53-57.

[Parnas69]D. L. Parnas, " On the Use of Transition Diagrams in the Design of A User Interface for an Interactive Computer System", *ACM National Conference*, 1969, pp.379-385.

[Pavlidis84]

T. Pavlidis, " PED: A 'Distributed' Graphics Editor", *Graphics Interface*, 1984, pp.75-79.

[Rogers81]G. T. Rogers and M. B. Feldman, An Intermediate language and an Interpreter for Style-Independent Interactive System", Report GWU-IIST-81-21, Institute for Information Science and Technology, Dept. of Electrical Engineering and Applied Science, The George Washington University, August, 1981.

[Simpson82]

H. Simpson, " A Human-Factors Style Guide for Program Design", *Byte* 7, 4 (April, 1982), pp.108-132.

[Singh83] B. Singh, J. C. Beatty and R. Rymann, " A Graphics Editor for Benesh Movement notation", *Computer Graphics*, July, 1983, pp.51-62.

[Smith82] D. C. Smith, C. Inby, R. Kimball and B. Verplank, " Designing the Star User Interface", *Byte* 7, 4 (April, 1982), pp.242-282.

[Tanner84]P. P. Tanner and W. A. S. Buxton, " Some Issues In Future User Interface Management System (UIMS) Development", *in [Seeheim84]*, 1984.

[Waervagen83]

T. Waervagen, F. Lillehagen and J. Losenedahl, " ICAN ICUE, ICAN's User Interface Management System", *UIM Workshop*, November, 1983.

[Wong82] P. C. S. Wong and E. R. Reid, " FLAIR-User Interface Dialog Design Tool", *Computer Graphics 16*, 3 (1982), pp.83.

Appendix A1

Example of Designing Dialogue Control

In this appendix an example of the use of our system to produce the dialogue control component of a user interface is presented.

1.1. Problem

The problem to be solved is a simple application in which two types of objects (circles and squares) can be edited on the screen. A menu is used to select the objects to be manipulated. By moving the tracking cross to the working window and pressing a button on the tablet, the program will display the object with the tracking cross positioned at the center of the object. An "EXIT" command is available in the menu to exit from the program.

1.2. Recursive Transition Network Solution

The main RTN for this problem consists of four states as shown in Figure A1.1. Three input tokens can be sent from the presentation component. They represent the three items in the menu. Whenever the user chooses one of them, the corresponding token will be sent to the dialogue control component which causes it to change state. The EXIT token causes the program to terminate (because it will go to an ending state), the other two tokens cause the program to call another subdiagram (either drawc or draws) to add the appropriate objects to the screen.

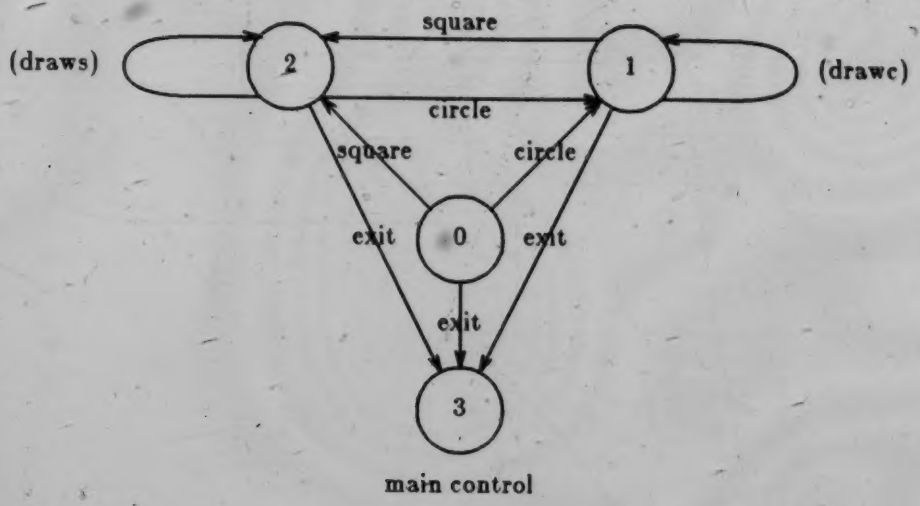

Figure A1.1 Main Control RTN

These two subdiagrams are shown in Figure A1.2 below. They are used to draw circles and squares respectively. "Point" is the input token that is sent from the presentation component when the user presses the button in the work area. Depending on the current state of the system (i.e. depending on whether "square" or "circle" has been chosen previously), either of them will be called when "point" is entered and the appropriate object is drawn.

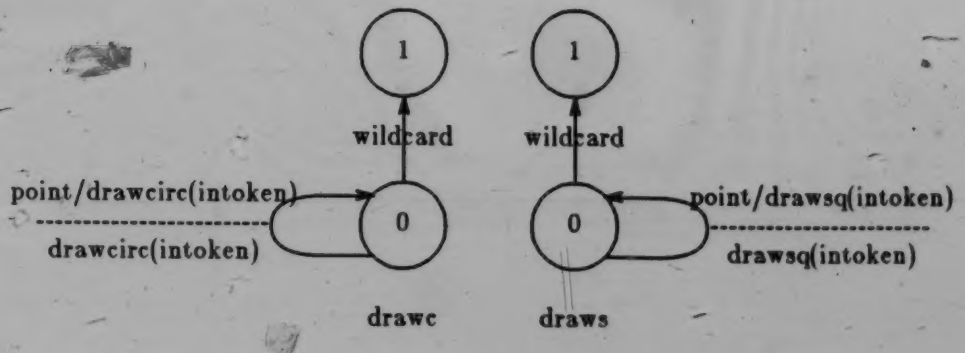

Figure A1.2 Two subdiagrams called by 'Main Control'

These subdiagrams send tokens to the presentation component and the applica-

tion interface module. The token sent to the presentation component is used to add the object to the screen; while the token for the application interface module updates the database.

1.3. Code Generation

Based on the three subdiagrams as shown in the previous section, three event handlers will be formed along with the three Leading Relation files. The contents of these files are:

Main control	Drawc	Draws
square	point	point
circle	exit	exit
exit		

The generated code for the three diagrams is:

Main Control Diagram:

```
ehmain
2 6 6
event0 event1 event2 continue event4
circle event0
square event1
exit event2
continue continue
point event4
%
ehmain ( iname, ename, value, variables )
 int iname;
 int ename;
 int value;
 int variables[ ];
{
 switch ( ename )
 {
  case ( event0 ):
   switch ( state )
   {
    case ( 0 ):
     state = 1;
     return( );
    case ( 2 ):
     state = 1;
     return( );
   }
  case ( event1 ):
   switch ( state )
   {
    case ( 0 ):
     state = 2;
     return( );
    case ( 1 ):
     state = 2;
```

```
        return( );
      }
    case ( event2 ) :
      switch ( state )
      {
        case ( 0 ):
          state = 3;
          send_event( variable[ 0 ], continue, variable[ 1 ] );
          send_event( variable[ 0 ], ename, 0 );
          destroy( iname );
          return( );
        case ( 1 ):
          state = 3;
          send_event( variable[ 0 ], continue, variable[ 1 ] );
          send_event( variable[ 0 ], ename, 0 );
          destroy( iname );
          return( );
        case ( 2 ):
          state = 3;
          send_event( variable[ 0 ], continue, variable[ 1 ] );
          send_event( variable[ 0 ], ename, 0 );
          destroy( iname );
          return( );
      }
    case ( continue ) :
      switch ( state )
      {
        case ( temp1 ):
          state = 1;
          return( );
        case ( temp2 ):
          state = 2;
          return( );
      }
    case ( event4 ) :
      switch ( state )
      {
        case ( 1 ):
          create_instance( drawc, 2, this_instance, temp1 );
          send_event( new_instance, ename, value );
          state = temp1;
          return( );
        case ( 2 ):
          create_instance( draws, 2, this_instance, temp2 );
          send_event( new_instance, ename, value );
          state = temp2;
          return( );
      }
  }
  switch ( state )
  {
    case ( 1 ):
      create_instance( drawc, 2, this_instance, temp1 );
      send_event( new_instance, ename, value );
      state = temp1;
      return( );
    case ( 2 ):
      create_instance( draws, 2, this_instance, temp2 );
      send_event( new_instance, ename, value );
      state = temp2;
      return( );
  }
}
%
```

Drawc Subdiagram:

```
ehdrawc
2 2 3
event1
point event1
%
ehdrawc ( iname, ename, value, variables )
 int iname;
 int ename;
 int value;
 int variables[ ];
{
 switch ( ename )
 {
  case ( event1 ) :
   switch ( state )
   {
    case ( 0 ):
     state = 0;
     send_token( 'APPLICATION, 1, drawcirc, intoken );
     send_token( PRESENTATION, 1, drawcirc, intoken );
     return( );
   }
 }
 switch ( state )
  {
    case ( 0 ):
     state = 1;
     send_event( variable[ 0 ], continue, variable[ 1 ] );
     send_event( variable[ 0 ], ename, 0 );
     destroy( iname );
     return( );
  }
}
%
```

Draws Subdiagram:

```
ehdraws
2 2 3
event1
point event1
%
ehdraws ( iname, ename, value, variables )
 int iname;
 int ename;
 int value;
 int variables[ ];
{
 switch ( ename )
 {
  case ( event1 ):
   switch ( state )
   {
    case ( 0 ):
     state = 0;
     send_token( APPLICATION, 1, drawsq, intoken );
     send_token( PRESENTATION, 1, drawsq, intoken );
     return( );
   }
 }
 switch ( state )
 {
    case ( 0 ):
     state = 1;
     send_event( variable[ 0 ], continue, variable[ 1 ] );
     send_event( variable[ 0 ], ename, 0 );
     destroy( iname );
     return( );
  }
}
%
```

Based on these generated code, the assembler and the compiler of the UIMS will form the tables and the "C" language event handlers.

Appendix A2

User Manual

Tools for Recursive Transition Networks

User's Manual

Version 1.0

Written by

Sai Choi Lau

Spring, 1985

Department of Computing Science

University of Alberta

Edmonton Alberta

Canada

2.1. Introduction

This document describes one of the tools used to specify the dialogue control component of the University of Alberta User Interface Management System. It describes all the available commands and how to use them.

2.2. Overview

The tools available in this system are aimed at helping the user interface designer (the editor user) to create and modify the dialogue control component of the U of A UIMS. Recursive Transition Networks (RTNs) are entered into the system in a graphical way and the output is an EBIF file describing the RTNs. This file is processed by an assembler and a "C" language compiler resulting in an object module that is linked with other modules (i.e. the presentation component and the application interface module) to form a complete user interface. The three tools available are:

(1) Graphical Editor.

An editor used to enter and modify the RTNs interactively and produces a database describing the diagrams. A datafile can also be generated by this editor to produce a hardcopy of the RTNs.

(2) Associated Facilities.

Procedures to manipulate the databases created by the graphical editor. The main operations are merging, destroying and copying the databases.

(3) Code Generator.

Program to convert the database of the RTNs into EBIF.

2.3. System configuration

The basic hardware requirements of the system are a colour monitor, a tablet with at least two buttons for selection, a VAX† computer or a similar machine that can run the UNIX‡ 4.2 BSD operating system and a laser printer (optional).

2.4. Graphical Editor

2.4.1. Using the Editor

The graphical editor can be called to create or edit the RTNs by:

ge network-name [-p] [-t]

Network-name is the name of the database to be called. If this database has previously been created, the first diagram on this database is loaded and displayed on the monitor. Otherwise, a new database is created with "network-name" as its name. The "-p" option is used to specify whether the user is an expert or novice. If this flag is set, the editor will not display any prompting messages and treats the user as an expert. "-t" option tells the editor to produce a file for the text-formatter "troff" in order to get a hardcopy of the RTN.

The primitives of the RTNs are the objects in the transition diagrams which are linked together to represent the dialogue. The four basic primitives are state, state name, arc and arc parameter. The arc parameter is divided into three different fields: input token, output token and application procedure. Each of these can be manipulated separately. Two other primitives, group and network, are used in this editor, They represent a group of primitives and the transition diagram that is being edited.

The editor commands are used to manipulate these primitives. The common commands are addition, deletion, removing, renaming, and "exit". Auxiliary com-

† VAX is a trademark of Digital Equipment Corporation.
‡ UNIX is a trademark of AT & T Bell Laboratories, Inc.

mands include "grouping", "next network" and "help". The meaning of these commands and their syntax is explained in the following sections.

2.4.2. Windows

The editor makes use of the screen to display all the necessary information and guide the user in entering and creating the recursive transition networks. Four windows are being used and their layout is shown in figure A2.1 below.

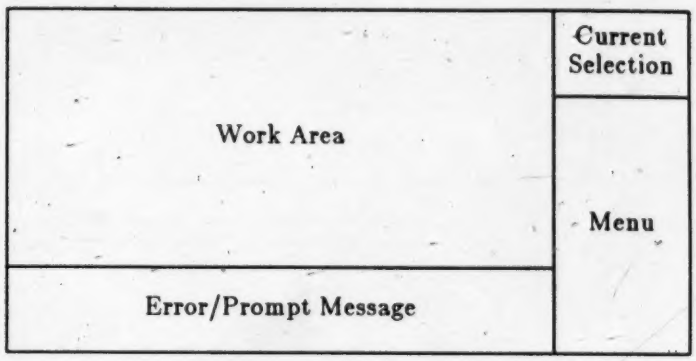

Figure A2.1 Window Layout

The work area is used to display the diagram being edited. This gives immediate feedback of what the user has entered. The error and/or prompt messages are shown in another window. It explains the error that the user has created and/or prompts the user to interact with the editor. Besides showing these messages, help messages for all the commands and primitives are also displayed on this window. The menu window shows the commands and primitives available for selection while the "Current Selection" window displays the choices the editor is using. The name of the transition diagram is shown on this window as well.

2.4.3. Selection of Commands and Primitives

Since the editor is menu driven all the commands and primitives are selected by putting the tracking cross on the desired menu item and pressing the PF1 key on the tablet †. The user can change the chosen options by selecting again and no ordering of primitive and command selection is required.

2.4.4. Error Handling

Whenever the editor detects that the user has entered an incorrect command, the error handling routine will be invoked. This procedure displays an error message explaining what is wrong with the input data and rings a bell to notify the user. Commands can then be re-entered or the user can ignore the current command by pressing the PF3 key and starting a new one.

2.5. Editor Primitives

The primitives are the objects on the recursive transition networks that are linked together to represent the dialogue sequence. The objects that are used in the editor include the state, the state name, the arc, the arc parameter, the input token, the output token, the application procedure token, the network id and the group. Each of these is described in the following sections.

2.5.1. State

State is used to represent a state of the recursive transition networks. In the diagram, it is represented by a fixed size circle. States can overlap each other but complete overlapping is not desired because in this case, there is no way to access the covered circle. This primitive can be chosen from the menu by putting the tracking cross in the item "STATE" and pressing the PF1 key.

---

† Any key that can generate event type 1 according to the WINDLIB specification can be used.

2.5.2. State Name

The purpose of assigning a state name to each state is to make it possible to refer to it textually. In the menu, it is represented by "STATE NAME". An integer is used as the state name and it is put at the center of the state.

2.5.3. Arc

An arc is used to connect two states together. This is used to represent a transition from one state to another. Four points are specified by the user for every arc and the resulting curve is a Bezier curve of the points. It is represented as "ARC" in the menu. The length, curvature and direction of the arc is fixed by the user and the curve can go through other primitives if required.

2.5.4. Arc Parameter

This parameter is associated with an arc. Whenever the arc is traversed, the arc parameter is processed. The user can choose this primitive by pressing the PF1 key at the option "ARC PARAMETER". Three fields are grouped together in this parameter, they are the input token field, the output token field and the application procedure field. The input token field holds the token sent from the presentation component or another subdiagram name. If the user enters a token which matches the input token, it will cause the arc to be traversed. If this field holds a diagram name, the corresponding diagram will be called and program control is passed to the called subdiagram if the token entered by the user satisfies the Leading Relation set of the called subdiagram. The output token is the token sent to the presentation component as a result of this arc traversal. The application procedure is the name of the token which is sent to the application interface model of the UIMS to call the application procedure.

Any of these three fields can be left blank and their names can be in any combina-

tions of characters and letters, underline is allowed as well. In the diagram, it is represented as $\frac{\textit{input token/output token}}{\textit{application procedure name}}$. If the output token or the procedure name is not specify, the representation changes to *input token/outputtoken* or $\frac{\textit{input token}}{\textit{application procedure name}}$ respectively.

2.5.5. Input Token, Output Token and Application Procedure

The input token, the output token and the application procedure can be edited individual and are treated as other primitives. They are shown as "INTOKEN TOKEN", "OUTTOKEN TOKEN" and "PROCNAME" in the menu.

2.5.6. Network ID

The network id is the name of the RTN being edited. Any combinations of letters, digits and underlines are allowed as the name. In the menu, it is shown as "NETWORK ID".

2.5.7. Group

Primitives can be collected together to form a group and a name is assigned to it. The name of a group can be of any combination of letters, digits and underlines. Once a group is formed, it can be used in the same way as other primitives. Groups should have different names in order to differentiate between them. The user can choose groups by selecting "GROUP" from the menu then the editor will prompt the user to enter the name of the group. The advantage of using groups is to speed up the operation since commands are applied to several objects at the same time instead of single primitives.

2.6. Editor Commands

2.6.1. Add

The "Add" command is used to add a primitive to the transition diagram and updates both the database and the display. The primitives that can be used with this command include the state, the state name, the arc, the arc parameter, the input token, the output token, the application procedure and the group.

"ADD STATE" will cause a circle representing the state to draw on the diagram. The user needs to specify the center of this circle on the work area by pressing the PF1 key when the tracking cross is at the desired location. After a state is created, a state name can be added by choosing the primitive "STATE NAME". The state which holds this name is specified by putting the tracking cross inside the desired state and then entering the state name from the keyboard.

Before the "ADD ARC" command is used, the states at the ends of the arc must exist. The user needs to enter 4 positions to fix the locus of the arc. The first two positions are inside the tail and the head states. The two other points are used to fix the locus. These points are joined together by a Bezier curve.

When the arc parameter is added, two states representing the tail and the head states of the arc must be specified. This is done by pressing the PF1 key on the tablet when the tracking cross is inside the states. Then the tracking cross is placed at the position at which the parameter is displayed †. The parameter is then entered in the format input-token/output-token/procedure-name. There may be cases in which more than one arc has the same tail and head states. In this situation, the arcs are blinked in turn and a prompt message is displayed asking the user to select the arc. Typing "y" will select the blinking arc, other keys will switch to blink another arc.

---

† No pressing of key is required.

"INTOKEN", "OUTTOKEN" and "APPL. PROCEDURE NAME" are the primitives used to add the input token, the output token and the application procedure name individually to an existing arc parameter with the respective field empty. In order to select the desired arc parameter, the user needs to place the tracking cross in the parameter (which is assumed to be a rectangle) and then enters the name. After the carriage return is hit, the name is inserted into the appropriate position.

A group of objects can be added by choosing "GROUP" as the primitive. The user specifies the name of the group to be added, this name is displayed on the screen automatically.

2.6.2. Delete

This command is used to delete a primitive from both the monitor and the database. The primitives that can be used with this command are the state, the state name, the arc, the arc parameter, the input token, the output token, the application procedure and the network id.

The primitive "STATE" is used to delete a state from the transition diagram. This will cause the state record to be removed from the database and the screen to be updated. If a state name has already been assigned to a state, it is deleted as well. The user just places the tracking cross inside a state and presses the PF1 key to execute this command.

In order to delete a state name from the transition diagram, the primitive "STATE NAME" is chosen. The tracking cross is placed inside the state for which its name is deleted and the PF1 key is pressed. The state itself is unaffected by this command.

An arc joining two states can be deleted by the primitive "ARC". The tail and the head states are specified by putting the tracking cross inside the circles and pressing the PF1 key. If an arc parameter is associated with the arc, it is deleted as well.

When there are two or more arcs connecting the same set of tail and head states, the arcs will blink alternative and a message will prompt the user to choose the appropriate arc. The user can type "y" to delete the blinking arc. If an other key is pressed, the next arc is blinked.

The arc parameter itself can also be deleted by selecting "ARC PARAMETER". This command is executed when the tracking cross is placed on the parameter and the PF1 key is pressed.

The input token, the output token and the application procedure name can also be detected by choosing the respective primitive. Their operation is the same as the "DELETE ARC PARAMETER".

A group of objects is deleted by using the "GROUP" primitive. The only operation required is to enter the name of the group when the editor prompts the user.

"NETWORK ID" is used to delete the current transition network from the database and the previous diagram is displayed. When this command is executed, the editor will ask for confirmation and "y" needs to be entered to confirm. Pressing any other key will ignore the command.

2.6.3. Move

The purpose of this command is to move a primitive from one part of the diagram to another part. This command can only be applied to primitives within the same diagram. The only primitives that can use this command are the state, the state name, the arc, the arc parameter, the input token, the output token and the application procedure.

When the primitive "STATE" is chosen, a state can be moved from one location to another. The user chooses the state by pressing the PF1 key when the tracking cross is inside the desired state. The intended location is selected by pressing the key

again to fix the new center. When the state is moved, the state name will move as well.

"STATE NAME" is the primitive used to move a state name from one state to another. The user first chooses the state name to be moved by pressing PF1 key inside the state. The intended state is selected by pressing the key again when the cross is inside the state.

An arc can be moved by selecting the tail and the head states and then the new set of states. The two intermediate points cannot be relocated. If there are two or more arcs connecting the same tail and head states, the arcs are blinking in turn to ask the user to choose the intended one. Pressing any key except "y" will cause the next arc to be displayed. When "y" is pressed, the editor will stop blinking and the blinked arc is used as the desired arc.

The primitive "ARC PARAMETER" is used to move the arc parameter from one location to another. The parameter to be moved is first selected by pressing the PF1 key when the tracking cross is on the parameter. The new tracking cross position is the new starting position of the input token field of the arc parameter.

"INTOKEN", "OUTTOKEN" AND "APPL. PROCEDURE NAME" are the primitives used to move the input token, the output token and the application procedure name from the original to a new position. The sequence for execution is the same as "Move Arc Parameter" with the new location specified for the input token, the output token and the application procedure name respectively.

2.6.4. Rename

"Rename" is used to rename a primitive on the current transition diagram. These primitives include the state name, the arc parameter, the input token, the output token, the application procedure and the network id.

The "STATE NAME" is the primitive used to rename the state. The user specifies the state by placing the tracking cross inside the state and then enters a new state name. The new name will replace the old one.

The parameters "ARC PARAMETER", "INTOKEN", "OUTTOKEN" AND "APPL. PROCEDURE NAME" are used to rename the arc parameter, the input token, the output token and the application procedure name. The user places the tracking cross over the intended parameter and enters the new name.

The diagram name being edited can also be changed by entering the primitive "NETWORK ID" and then entering the new name from the keyboard.

2.6.5. Next Network

The "NEXT NETWORK" command is used to finish editing the current diagram and load or create another diagram for editing. No primitive is needed for this command and once the command is selected, the editor will ask the user for the new diagram name. If this name is in the database, the diagram is loaded and displayed on screen; otherwise a new entry is created in database and the screen is cleared to indicate a new database entry.

2.6.6. Add Group

Two commands are available to form groups. "ADD GROUP BY LOCATION" and "ADD GROUP BY POINTING" are used to collect objects together. The only difference between them is the way in which the objects are grouped. Both commands need to specify the name of the group first, but the first command collects objects by specifying the lower left and upper right corner of a rectangle. All objects which are completely inside the rectangle are included in the group. The second command selects the objects by using the tracking cross for selection. The selection method for various objects are the same as the "DELETE" command and the group list is terminated by pressing carriage return.

2.6.7. Merge Groups

This command is used to merge two groups. When this command is issued, the editor will ask the user to enter the two group names. The two names are separated by spaces and the first name is the group to be merged and the resulting group is stored in the second name. The first group is unaffected by this command.

2.6.8. List Group

When "LIST GROUP" is selected, the editor will ask the user to enter the group name to be listed. If this name exists in the database, the collected objects are displayed. If the user enters "all", all the groups in the database will be displayed. If "all" is used, all the objects in the first group are displayed and the editor asks the user to press the PF1 key to continue displaying the second group and so on. Pressing any other key will quit this command.

2.6.9. Rename Group

The "RENAME GROUP" command is used to rename an existing group. The user is asked to enter the old and new group names separated by spaces and terminated with a carriage return.

2.6.10. Destroy Group

The "DESTROY GROUP" command is used to destroy a group from the group database. A group name is entered by the user for this command.

2.6.11. Help

The "HELP" command is used to provide on-line help messages for the editor commands and primitives. This command is on all the menus and once it is selected, the user needs to enter the command for help. Only the commands or primitives that are shown on the current menu can call for explanation.

2.7. Associated Facilities

2.7.1. Merge Databases

This command is used to merge two databases created by the editor together. The syntax of the command is

merge from-db to-db

If either or both of the databases do not exist, an error message is printed and the command is not executed. "from-db" is the name of the database that is merged with the other database. "to-db" is the name of the resulting database. The data in "from-db" is unaffected by this command.

2.7.2. Copy Database

A database can be copied to create another duplicate database by using the command

copy from-db to-db

"from-db" is the name of the database to be copied and "to-db" is the name of the duplicated version.

2.7.3. Destroy Database

The command used to destroy an existing RTN database is

destroy db-name

"db-name" is the name of the database to be destroyed. If this database does not exist, an error message is printed. No confirmation is used in this command.

2.8. Code Generation

After a RTN database is produced by the graphical editor, the code generator can be used to generate the EBIF of the dialogue. The command is:

convert db-name [db-name...]

Two types of files are created by this command, the Leading Relation files of all the diagrams in the databases and their FBIF files. The later ones are the essential files to form a complete user interface.

The name of the event handler is the same name as the RTN with "eh" appended at the beginning. The Leading Relation files use the same name as the event handler files with ".L" added at the end. Each database contains a number of diagrams and each of them will create a separate set of files.

"db-name" is the name of the database that is converted to event based form. Any number of databases can be converted at the same time. If any of these databases do not exist, the command is exited with an error message. No event based format

files is created but Leading Relation files for all the diagrams in the existing databases up to the place where the error is detected are created.